

User Manual for



Mattias Blennow
and
Enrique Fernandez-Martinez

Department of Theoretical Physics, KTH Royal Institute of Technology
Roslagstullsbacken 21, 106 91 Stockholm, Sweden

Departamento de Física Teórica UAM
Cantoblanco, 28049 Madrid, Spain

Version 1.3.0

Copyright © 2008-2013 Mattias Blennow, Enrique Fernandez-Martinez.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Abstract

MonteCUBES is a package created to provide the possibility of easily performing Markov Chain Monte Carlo (MCMC) simulations of neutrino oscillation experiments. This document is the end-user manual for **MonteCUBES** and thus provides instructions on how to install and use it along with other tools. The **MonteCUBES** distribution consists of two main parts. The first part of the distribution is a C library written as a plug-in to **GLOBES** [1, 2]. This part enables the user to perform MCMC samplings of the parameter space of neutrino oscillations. The fact that it is written as a plug-in to **GLOBES** makes it simple to define and study different experimental setups, as well as the possibility of adding different new physics. The second part of **MonteCUBES** is a graphical **Matlab** interface, which makes it easy to plot simulation results in a variety of different ways, as well as export higher-level data such as contours and graphs, without reference to the originally produced MCMC results.

Terms of usage

Referencing the MonteCUBES software

MonteCUBES has been developed as a tool for academic use. Thus, the authors of MonteCUBES would appreciate to be given proper academic credit when it is used. If you use MonteCUBES to produce a publication or talk, please cite the following reference:

M. Blennow and E. Fernandez-Martinez
Neutrino oscillation parameter sampling with MonteCUBES
arXiv:0903.3985 [hep-ph]

Do *not* cite this manual itself, it is not a scientific publication and will evolve along with the MonteCUBES software. Of course, if you are using MonteCUBES then you will also be using GLoBES. Do not forget to give proper academic credit also to the GLoBES developers (see the GLoBES manual for details).

Apart from the above, MonteCUBES is free and open-sourced software, distributed under the GNU General Public License.

Contents

1	Introduction	1
1.1	What is MonteCUBES?	1
1.2	What is MonteCUBES not?	2
1.3	Versioning	2
2	The MonteCUBES C library	5
2.1	Installation	5
2.1.1	Prerequisites for installation of MonteCUBES	5
2.1.2	Installation Instructions	5
2.1.3	Basic Installation	6
2.1.4	If your GLoBES copy is installed in a different directory	7
2.2	API definitions	7
2.2.1	Basic usage definitions	8
2.2.2	Advanced usage definitions	12
2.3	Definitions of outfiles	21
2.3.1	Summary files	21
2.3.2	Chain files	22
2.4	Examples	23
2.4.1	Simulation of the ISS neutrino factory	23
2.4.2	Treating degeneracies	28
3	The MonteCUBES Matlab GUI	31
3.1	Starting the GUI and reading simulation results	31
3.2	Making plots	33
3.2.1	1D histogram plots	36
3.2.2	1D chain progression plots	37
3.2.3	1D confidence region plots	38
3.2.4	2D scatter plots	39
3.2.5	2D counts contour plots	39
3.2.6	Triangle plots	42
3.2.7	3D surface plots	44
3.3	Results from the degenerate solution simulation	45
3.4	Examples of bad convergence	46
3.5	Notes on filtering	46

A	Markov Chain Monte Carlo theory	51
A.1	The Metropolis–Hastings sampling algorithm	51
A.1.1	Convergence and finer details	52
A.2	Usage and interpretation of GLoBES methods	54
A.2.1	The GLoBES χ^2 as the log-likelihood	55
A.2.2	Notes on priors	56
A.2.3	Interpreting the results	56
A.3	The MonteCUBES degeneracy solver	57
A.3.1	Chain heating	58
A.3.2	Degenerate steps	58
A.3.3	Which method do I use?	59
A.3.4	The temperature lowering degeneracy finder	59
B	The NonUnitarity Engine (NUE)	63
B.1	Theory of a non-unitary mixing matrix	63
B.2	API definitions	64
B.2.1	Methods	64
B.2.2	Constants	70
C	The nSI Event Generator Engine (nSIEGE)	71
C.1	Theory of a non-standard matter interactions	71
C.2	API definitions	72
C.2.1	Methods	72
C.2.2	Constants	78
D	GNU General Public License	81
E	GNU Free Documentation License	95
1.	APPLICABILITY AND DEFINITIONS	95
2.	VERBATIM COPYING	97
3.	COPYING IN QUANTITY	97
4.	MODIFICATIONS	98
5.	COMBINING DOCUMENTS	100
6.	COLLECTIONS OF DOCUMENTS	100
7.	AGGREGATION WITH INDEPENDENT WORKS	101
8.	TRANSLATION	101
9.	TERMINATION	101
10.	FUTURE REVISIONS OF THIS LICENSE	102
11.	RELICENSING	102
	ADDENDUM: How to use this License for your documents	103

Chapter 1

Introduction

1.1 What is MonteCUBES?

The Monte Carlo Utility Based Experiment Simulator (**MonteCUBES**) is a utility designed to provide a simple and straightforward way of performing Markov Chain Monte Carlo (MCMC) sampling of the parameter space for any setup of terrestrial neutrino oscillation experiments. In particular, it has lately become increasingly popular to not only consider neutrino oscillations within the standard framework, in which the parameters consist of two mass square differences, three mixing angles and a CP -violating phase, but also to include the effects of various non-standard physics. Thus, the parameter space is in general high-dimensional and this makes deterministic algorithms for probing it, such as gridding and deterministic minimization, inefficient. Instead, the complexity of a stochastic algorithm does not grow at the same rate with dimensionality and it is therefore ideal for problems of this kind. The **MonteCUBES** distribution consists of two parts:

1. A C library in the form of a plug-in to the General Long Baseline Experiment Simulator (**GLOBES**) [1, 2]. As the **GLOBES** software package already includes several methods for defining neutrino oscillation experiments, computing event rates, etc., there would be no use in doubling this work. Thus, the **MonteCUBES** C library makes use of these methods in order to provide a simple and structured user interface. It adds methods for defining MCMC parameters and for performing the actual simulation. The actual Monte Carlo method employed is the Metropolis–Hastings sampling algorithm (see App. A). Since the **MonteCUBES** C library is a plug-in to **GLOBES**, files used with **GLOBES** for describing experiments (Abstract Experiment Definition Language, AEDL, files) will also work with **MonteCUBES**. The **MonteCUBES** C library was constructed with **GLOBES** 3.0, but should be compatible also with some older versions of **GLOBES**. However, **GLOBES** 3.0 is the first version implementing the possibility of defining new physics and one of the major strengths

of **MonteCUBES** is the possibility of exploring large-dimensional parameter spaces.

2. *A graphical user interface (GUI) for Matlab.* The raw sampling data produced by the **MonteCUBES** C library can, although it is well-defined, be somewhat cumbersome to handle. In order to facilitate the usage of **MonteCUBES**, its distribution also includes a set of **Matlab** files intended to simplify this process and create results which are easier to overview and interpret. The GUI will read the raw sampling data and provide the means to produce a number of different plots using that data. In addition to allowing the data to be plotted in **Matlab**, the GUI also provides the possibility of exporting higher level data (such as actual contours and tabulized functions rather than the raw samples) for inclusion in a plotting program of the user's choice. No preknowledge of **Matlab** is required in order to use the GUI, although some experience in writing inline functions may be helpful for some of the more advanced features.

1.2 What is MonteCUBES not?

MonteCUBES is *not* a stand-alone application. Thus, it is highly dependent on other software. In particular, since the C library is a **GLOBES** plug-in, a working installation of **GLOBES** is required in order to use it (we recommend version 3.0 or later). However, apart from the dependency on **GLOBES**, the **MonteCUBES** C library does not have any additional requirements over those already put by **GLOBES** (such as the proper compilers and scientific libraries).

Naturally, the **Matlab** GUI also requires a working copy of **Matlab**. The GUI was written for **Matlab** 7.0 and earlier versions are not supported (although the GUI might still work properly). However, processing the raw sampling files created by the C library is possible even if the C library is not installed.

1.3 Versioning

The current version of **MonteCUBES** is 1.3.0. The version number has the following interpretation:

- The first number denotes the release number. It will increase by one when major updates, rewriting of code, and new functions are added compared to the previous release.
- The second number denotes the major revision number. It is increased by one whenever new functionality is added.
- The final number denotes the minor revision number. It is increased whenever a new distribution is put together after any minor change has been made.

- In addition, the version number may be followed by a **b**, denoting that it is a beta version. In this case, the distribution has been released for public testing, but it is still experimental and more likely to contain bugs.

Our aim is to improve on **MonteCUBES** whenever possible. If you want to give us feedback, please do so at **emb@kth.se** / **enrique.fernandez-martinez@uam.es**. In order for us to properly address your feedback, we would appreciate if you would include information which versions of **MonteCUBES**, **GLOBES**, and **Matlab** you are using.

Chapter 2

The MonteCUBES C library

2.1 Installation

2.1.1 Prerequisites for installation of MonteCUBES

Installation of MonteCUBES requires that you have a working installation of GLoBES. GLoBES can be downloaded from:

`http://www.mpi-hd.mpg.de/lin/globes/`

MonteCUBES was developed with GLoBES 3.0.12. GLoBES versions earlier than 3.0 are not supported by MonteCUBES.

The MonteCUBES installer will install the library `libmontecubes` into the same directory as your GLoBES installation. In order to compile programs using this library, you will therefore only have to add `-lmontecubes` to the linker options.

2.1.2 Installation Instructions

MonteCUBES follows the standard GNU installation procedure with the additional requirement that you must install it in the same directory as GLoBES. To compile MonteCUBES you will need an ANSI C-compiler. After unpacking the distribution the Makefiles can be prepared using the configure command,

`./configure`

NOTE! If you did not install GLoBES in the default directory, then you will have to add the option `--prefix=GLB_DIR`, where `GLB_DIR` is the absolute path to your GLoBES installation.

You can then build the library by typing,

`make`

A shared version of the library will be compiled by default.

The MonteCUBES library can then be installed using the command,

```
make install
```

The default install directory prefix is `/usr/local`.

2.1.3 Basic Installation

These are generic installation instructions.

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile` in each directory of the package. It may also create one or more `.h` files containing system-dependent definitions. Finally, it creates a shell script `config.status` that you can run in the future to recreate the current configuration, a file `config.cache` that saves the results of its tests to speed up reconfiguring, and a file `config.log` containing compiler output (useful mainly for debugging `configure`).

If you need to do unusual things to compile the package, please try to figure out how `configure` could check whether to do them, and mail diffs or instructions to the address given in the `README` so they can be considered for the next release. If at some point `config.cache` contains results you don't want to keep, you may remove or edit it.

The file `configure.ac` is used to create `configure` by a program called `autoconf`. You only need `configure.ac` if you want to change it or regenerate `configure` using a newer version of `autoconf`.

The simplest way to compile this package is:

1. `cd` to the directory containing the package's source code and type `./configure` to configure the package for your system. If you're using `csh` on an old version of System V, you might need to type `sh ./configure` instead to prevent `csh` from trying to execute `configure` itself.

Running `configure` takes a while. While running, it prints some messages telling which features it is checking for.

2. Type `make` to compile the package.
3. Type `make install` to install the programs and any data files and documentation.
4. You can remove the program binaries and object files from the source code directory by typing `make clean`. To also remove the files that `configure` created (so you can compile the package for a different kind of computer), type `make distclean`. There is also a `make maintainer-clean` target, but that is intended mainly for the package's developers. If you use it, you may have to get all sorts of other programs in order to regenerate files that came with the distribution.
5. Since you've installed a library don't forget to run `ldconfig`!

2.1.4 If your GLoBES copy is installed in a different directory

If you installed GLoBES in a directory different from the default (one reason for doing this could be lack of root privileges), then you will have to install MonteCUBES in that same directory as well. This is done by

```
./configure --prefix=GLB_DIR
```

where GLB_DIR is the directory of the GLoBES installation, and then follow the usual installation guide. Since MonteCUBES will then be installed in the same directory as GLoBES, you will not have to tell the compiler to add additional directories to look for header files. However, you will have to tell the linker that it should include the montecubes library.

A typical compiler command is

```
gcc -c my_program.c -I GLB_DIR/include/
```

and a typical linker command is

```
gcc my_program.o -lglobes -lmontecubes -L GLB_DIR/lib/ -o  
my_executable
```

More information on this issue can be obtained by having a look into the output of `make install`.

CAVEAT: It is in principle possible to have many installations on one machine, especially the situation of having an installation by root and by a user at the same time might occur. However it is strictly warned against this possibility since it is **extremely** likely to create some versioning problem at some time!

Installation Names

By default, `make install` will install the package's files in `/usr/local/bin`, `/usr/local/include`, etc. You can specify an installation prefix other than `/usr/local` by giving `configure` the option `--prefix=PATH`. The path into which you install MonteCUBES should be the same as the one where you installed GLoBES, *i.e.*, you should give the same options to the MonteCUBES configure script as you gave the GLoBES configure script.

2.2 API definitions

This section lists the functions defined in `montecubes.h` and specifies their usage. In order to use them, you must include `montecubes.h` as a header file in your program (typically by putting `#include <montecubes/montecubes.h>` along with the inclusion of the other headers). The header file is located in `$INSTALLDIR/include/montecubes` after installation (where `$INSTALLDIR` is the directory where you installed GLoBES and MonteCUBES).

2.2.1 Basic usage definitions

The functions listed below are the ones necessary to implement the most basic usage of `MonteCUBES`. Unless you are planning to customize your `MonteCUBES` usage by changing the step proposal function, prior function, etc., and only use the basic `MonteCUBES` methods, such as running a Markov Chain Monte Carlo simulation with Gaussian steps and no tweaks, these are the only functions you will need.

`int mcb_setChainNo (int N)`

This function sets the number of chains to use in the Monte Carlo. It has protections against setting a non-positive number. Use this function prior to calling the Monte Carlo method. The preset number of chains is four.

N – The number of chains to use.

Returns: `MCB_OK` if successful, `MCB_SET_ERR` otherwise.

`int mcb_setBurnNo (int N)`

This function sets the number of samples in the burn-in (per chain). These samples will not be considered when testing for convergence but will still be written to the result files. The burn-in length will be stored in the summary file so that the burned samples can be removed from the result if desirable. If the burn length is set to `MCB_DYNAMIC_BURN`, then a dynamic burn process will be used. The preset burn-in length is 1000.

N – The number of samples to use as burn-in.

Returns: `MCB_OK` if successful, `MCB_SET_ERR` otherwise.

`int mcb_setLengthMax (int N)`

This function sets the maximum length of one Markov Chain (after burn-in). This number of samples will never be exceeded even if the Monte Carlo has not converged. If no convergence check is made, then this number of samples will be produced. The preset maximum chain length is 10^6 .

N – The number to use for maximum chain length.

Returns: `MCB_OK` if successful, `MCB_SET_ERR` if input is smaller than minimum chain length.

`int mcb_setLengthMin (int N)`

This function sets the minimum length of one Markov Chain (after burn-in). This number of samples will always be produced, even if the Monte Carlo has converged. The preset minimum chain length is 5000.

N – The number to use for minimum chain length.

Returns: `MCB_OK` if successful, `MCB_SET_ERR` if input is larger than maximum chain length or input is negative.

`int mcb_setLengthMinMax (int Min, int Max)`

This function sets both the minimum and maximum length of one Markov Chain (after burn-in). The Monte Carlo will always produce a number of samples between these two numbers regardless of convergence checks. If no convergence check is made, then the chain length will reach the maximum number of samples. The preset minimum chain length is 5000 and the preset maximum chain length is 10^6 .

Min – The number to use for minimum chain length.

Max – The number to use for maximum chain length.

Returns: `MCB_OK` if successful, `MCB_SET_ERR` if input is ambiguous.

`int mcb_setConvergenceCriteria (glb_params r)`

This function sets the convergence criteria to use for each parameter. The Monte Carlo will be considered to have converged when $R - 1 < r$ for all parameters (see App. A). The default value for each convergence criteria is 0.05.

r – A GLOBES parameter vector containing the convergence criteria for each parameter

Returns: `MCB_OK` if successful, `MCB_SET_ERR` if invalid convergence criteria are given.

`int mcb_setConvergenceCheck (int N)`

This function sets how often the Monte Carlo will check for convergence. The convergence check will be performed each time this number of new samples has been produced. The default value for the convergence check is `MCB_CONV_CHECK_ON`, *i.e.*, convergence is not checked.

N – The number of samples produced between each convergence check. Use `MCB_NO_CONV_CHECK` if convergence should not be checked. The default, `MCB_CONV_CHECK_ON` is a large number, mainly intended to turn the convergence checks on when dynamic burning is used.

Returns: `MCB_OK` if successful, `MCB_SET_ERR` if invalid argument is passed.

`int mcb_setVerbosity (int N)`

This function sets when MonteCUBES will give feedback to `stdout`. Depending on the value set, the user will get different amounts of feedback (see below).

N – The verbosity level to use according to the following table:

- 0 or `MCB_NO_FEEDBACK`: At this level, `MonteCUBES` gives no feedback to the user.
- 1 or `MCB_ERROR_MESSAGES`: This level prints error messages to the screen whenever `MonteCUBES` discovers something which it is not able to perform (such as setting the maximum number of samples smaller than the minimum number of samples).
- 2 or `MCB_PROGRESS_BARS`: With this verbosity level, `MonteCUBES` will provide the user with progress bars displaying the progress of various tasks. It will also display the results of convergence checks.
- 3 or `MCB_SIMULATION_INFO`: This verbosity level will also make `MonteCUBES` print various information about the current simulation. This includes information on what it is doing and what is going on in the simulation.
- 4 or `MCB_WARNINGS`: The highest level of verbosity. This will display warning messages when the simulations are behaving in a strange way or the user sets strange (but valid) parameter values.

In addition to the feedback described in these verbosity levels, `MonteCUBES` will also display all messages from lower verbosity levels.

Returns:

```
int mcb_setVarName (int N, const char* newName)
```

This function sets the variable name for variable number `N`. The variable name is used in feedback to the user as well as in the output summary file. The variable names will appear and be used in the `MonteCUBES Matlab GUI`. The default setting is that the names of the standard parameters is set to a `TEX` code describing that parameter and that possible extra parameters are named `Extra parameter <k>`, where `<k>` is an integer.

`N` – The internal integer referring to the parameter whose name should be set. For example, this would be `GLB_THETA_13` if the user wants to set the variable name of θ_{13}

`newName` – A string containing the new name for the parameter.

Returns:

`MCB_OK` if the new variable name is set properly. `MCB_SET_ERR` if the new variable name is too long to fit into the buffers assigned for keeping track of the variable names. `MCB_ALLOC_ERR` if memory could not be allocated to store the variable name.

```
int mcb_setTemperature (double newT)
```

This function sets the temperature to be used in the MCMC simulations. The chains will have to be cooled in order to provide a sample of the true distribution if `newT` is different from 1.0. The preset value of the temperature is 1.0.

`newT` – The new temperature to be used in subsequent simulations.

Returns:

MCB_OK if the temperature is set. MCB_SET_ERR if `newT` is not positive. In addition, unless feedback is turned off, this function will give a warning if the temperature is set to less than one.

```
int mcb_setStepSizes (const glb_params step)
```

This function sets the typical step size when performing the Markov Chain Monte Carlo. The Monte Carlo will generate Gaussian steps with standard deviations given by the step sizes.

`step` – The step sizes to use in the Monte Carlo simulation.

Returns:

MCB_OK, since the parameter vector will simply be copied.

```
int mcb_addStartPosition (glb_params s)
```

This function should be used to add starting positions in parameter space to the Monte Carlo simulation. The actual starting points used will be generated through a step with stepsize three (3) away from the specified starting positions. This function can be called repeatedly. If more than one start position is added, the starting points of different chains will alternate between them. This method must be called at least once before running a MCMC simulation.

`s` – The parameter vector containing the starting position to be used.

Returns:

MCB_OK if successful. MCB_SET_ERR if the maximum number of starting positions has already been reached.

```
int mcb_clearStartPositions ()
```

This function clears all of the currently stored starting positions from memory. If called, new starting positions will have to be added before a MCMC simulation can be performed.

Returns:

MCB_OK

`int mcb_MCMC (char* outfile, int EXP, int RULE)`

This is the core function of `MonteCUBES`. It runs the MCMC and produces output files containing the resulting samplings. The MCMC parameters as well as the `GLOBES` initiations should be set to the desired values before calling this function.

`outfile` – A string containing the name of the outfiles to produce excluding the suffix. If set to "out", the outfiles will be named `out.mcb`, `out.mc1`, `out.mc2`, etc.

`EXP` – Which experiment to use according to the `GLOBES` standard. Use `GLB_ALL` for all experiments. This argument is simply passed to the `GLOBES` χ^2 function.

`RULE` – Which rule to use according to the `GLOBES` standard. Use `GLB_ALL` for all rules. This argument is simply passed to the `GLOBES` χ^2 function.

Returns:

`MCB_OK` if executed without problems, `MCB_N_ERR` if the MCMC has not converged, and `MCB_SET_ERR` if the step size or starting positions have not been set. `MCB_ALLOC_ERR` if at some point memory could not be allocated properly.

2.2.2 Advanced usage definitions

`MonteCUBES` includes several ways of customizing its behavior. Some of these features, such as customizing the way the Metropolis–Hastings algorithm is implemented in the simulation, are beyond the basic usage of `MonteCUBES`. Nevertheless, the advanced user may want to make these implementations. Thus, `MonteCUBES` also includes the possibility to do so. The below definitions will allow the user to fully customize the Monte Carlo Markov Chain simulations.

`void mcb_toPhysicalRegion (glb_params p, void* udata)`

This is the standard `MonteCUBES` function used to transform an arbitrary set of parameters to the physical region in such a way that the same set of parameters is always used to parametrize the same physical point in parameter space. The method is using setting C3 from Ref. [3] for the definition of the physical region.

`p` – The `GLOBES` parameter vector to transform to the physical region.

`udata` – Not used. No user data input required, passing `NULL` is sufficient. The argument is provided to comply with the standard for the function `mcb_setPhysicalTransformation`.

Returns: void

```
int mcb_setPhysicalTransformation (void(*transf) (glb_params,
void*), void* udata)
```

This function defines which transformation that should be used in order to transform a given set of parameters into the physical region. If the user does not set this, or if set to NULL, MonteCUBES will use `mcb_toPhysicalRegion`.

transf – The transformation used to transform a general `glb_params` structure into one where the parameters are in the physical region. The first argument is the set of parameters to transform. The second argument is a void pointer which can contain user specified data.

udata – The data to be passed as the second argument to **transf** whenever it is called.

Returns:

MCB_OK as it will always be possible to set the function and user data pointers. It is up to the user that the transformation works properly.

```
double mcb_standard_prior (const glb_params in, void* udata)
```

This function is essentially the same as the standard prior function in GLoBES. It adds priors based on which parameters that are left free according to the current projection. If the input errors have been set to less than 10^{-12} , then the method does not add a prior. *Note!* If you are getting `nan` as a prior value, then you have most probably forgotten to set the density parameters for one or more of **in**, the central values, or the input errors.

in – The parameter vector for which to compute the prior.

udata – Not used. Provided for compatibility with prior setter functions.

Returns: The prior value at **in**.

```
int mcb_setPriorFunction (double (*prior)(glb_params,void*), void*
udata)
```

This function allows the user to set a customized prior function, much like the feature that was introduced into GLoBES 3.0. It is introduced since GLoBES does not allow direct access to the prior function and all GLoBES methods actually using it are minimizers. The value of the prior function will be added to the χ^2 computed with systematics only. If not set before running the MCMC, then Gaussian priors will be used for the free parameters.

prior – The function to use as the prior. Its first argument should be the point in parameter space for which to compute the prior. The second argument is some user data that should be passed along to the prior function.

udata – This is the data that will be passed on to the prior function. This construction avoids the usage of global variables.

Returns:

MCB_OK as it will always be possible to set the function and user data pointers. It is up to the user that the prior function works properly.

```
int mcb_setRandGenerator (double(*randgen)(void*),void* udata)
```

This function sets the random generator used by MonteCUBES. By default, MonteCUBES uses a Mersenne twister algorithm. This function provides the possibility of changing this setting.

randgen – The random number generator which should be used by MonteCUBES. This should be a random number generator which returns a random number which is evenly distributed between zero and one.

udata – This is a pointer to data that the user wants to pass on to the random number generator. It will be passed as the argument of **randgen** when MonteCUBES is generating a random number.

Returns:

MCB_OK as it will always be possible to set the function and user data pointers. It is up to the user that the random number generator works properly.

```
int mcb_setRandSeed (int s)
```

This function allows the user to define the seed for the standard random number generator. If the user has not called this function when a simulation is run, the current time will be used as the seed.

s – The seed to use.

Returns:

MCB_OK as it will always be possible to set the seed to a given integer.

```
int mcb_addDegeneracyStep (glb_params step)
```

If degeneracies are expected, this function can be used in order to set the expected distance between two degeneracies. With a given probability,

the Monte Carlo will then add or subtract this step when computing the test steps so that it is possible to jump between the degeneracies and sample them with the correct weights. See the section on solving degeneracies in App. A for details.

step – The difference vector in parameter space between the degeneracies.

Returns:

MCB_OK if successful. MCB_SET_ERR if the maximum number of degeneracy steps has been reached.

```
int mcb_clearDegeneracySteps ()
```

This function clears all the degeneracy steps from memory. This can be useful if running several simulations in the same program and different degeneracies are present in the different simulations.

Returns:

MCB_OK as it is always possible to clear the list of degeneracy steps.

```
void mcb_setDegeneracySteps (glb_params* p, int* ind, int N)
```

This method can be used to set the appropriate degeneracy steps for a given set of degenerate solutions.

p – A `glb_params` array containing the degenerate solutions.

ind – An integer array containing the indices of **p** where the degenerate solutions are located. If all of the entries in **p** should be used, then `ind[k]` is equal to `k`.

N – The total number of degenerate solutions to set (*i.e.*, the length of `ind`).

Returns:

void

```
int mcb_readDegeneracySteps (char* file)
```

This method can be used to read the output file of the degeneracy locator (containing the degenerate solutions). It then sets the appropriate degeneracy steps for these solutions using `mcb_setDegeneracySteps`.

file – The name of the file to read. It should be the output of the degeneracy locator.

Returns:

MCB_OK if the degeneracy steps are set properly. MCB_IO_ERR if the file could not be read. MCB_SET_ERR if there were too many degeneracies to fit the steps into the memory allocated to degeneracy steps by MonteCUBES. MCB_ALLOC_ERR if the memory to store the degeneracy steps could not be allocated.

```
int mcb_TdegFinder(double Th, double Tl, int NT, int Nsamp, int
    exper, int rule, double maxChi2, int set, char* outfile)
```

This is the degeneracy locator currently implemented in MonteCUBES. It scans the parameter space for degeneracies by starting a number of chains at a high temperature and then gradually lowering the temperature (see App. A.3 for details).

Th – The starting temperature. Should be set to a sufficiently high value so that the degeneracies are merged.

Tl – The end temperature. Should be set to a low value so that the chains settle into local minima.

NT – The total number of temperatures to run the chains at.

Nsamp – The total number of chains to run. This should be fairly large to decrease the probability of missing a degenerate solution.

exper – The experiment for which to find the degeneracies. Passed on to the computation of the χ^2 .

rule – The rule for which to find the degeneracies. Passed on to the computation of the χ^2 .

maxChi2 – The maximal χ^2 difference for which to include degeneracies. Final results χ^2 larger than $\chi_{\min}^2$ plus this value will be ignored.

set – Flag for declaring if the finder should also set the degeneracy steps using `mcb_setDegeneracySteps`. Set to 0 if the steps should not be set and 1 if they should.

outfile – The name of the file in which to store the final results. Only results belonging to different local minima will be stored.

Returns:

MCB_OK if run properly. MCB_SET_ERR if the step sizes or starting positions are not properly defined. MCB_ALLOC_ERR if at some point memory could not be allocated properly.

```
int mcb_setDegeneracyStepProbability (double p)
```

This function sets the probability with which a degeneracy step is taken. The probability is the total probability of taking *any* of the stored degeneracy steps, *i.e.*, if there are n degeneracy steps and the probability is p , then the probability for each step is p/n .

p – The probability of taking a degeneracy step.

Returns:

MCB_OK if the probability is set. MCB_SET_ERR if p is not between 0 and 1.

```
int mcb_setStepProposalFunction (double (*proposal)(const glb_params,
const glb_params, const int*, const int*, int, int, double*, double,
glb_params, void*), void* udata)
```

This function sets the function that determines what test step to use in the Monte Carlo based on the current step and other simulation parameters. The function is set to `proposal`, which must be a function taking ten (10) arguments as follows (see App. A.1 for the Monte Carlo theory notation):

1. `const glb_params`: The current step x_n in the simulation.
2. `const glb_params`: The typical step sizes to use in the simulation, *i.e.*, the ones set by `mcb_setStepSizes`.
3. `const int*`: An integer array with flags notifying if a parameter should be free in the simulation or not (1 for free, 0 for fixed).
4. `const int*`: An integer array with flags notifying if a density should be free in the simulation or not (1 for free, 0 for fixed).
5. `int`: The number of free parameters N_f in the simulation (*i.e.*, the number of ones in argument 3).
6. `int`: The number of free densities N_d in the simulation (*i.e.*, the number of ones in argument 4).
7. `double*`: A double array containing Gaussian random numbers with variance 1. The size of the array is $N_f + N_d$.
8. `double`: A parameter denoting the relative step size. This will be one in the simulation, but the starting samples in the simulation will be determined from taking a step with a larger step size. In addition, the temperature lowering degeneracy locator uses different step sizes for different temperatures.
9. `glb_params`: This is where the new proposed sample $\hat{x}$ should be stored.
10. `void*`: This argument can contain user defined data. The `user_data` passed to `mcb_setStepProposalFunction` is passed in this argument.

Note that the user does not need to use all of these arguments in the function. For example, if the user wants to construct a simulation using transition functions which do not depend on the current step, argument 1 will not be used. However, the definition of the function should have this structure.

proposal – The function to use in order to generate new test steps.

udata – User defined data to be passed along to **proposal**.

Returns:

MCB_OK as it will always be possible to set the function and user data pointers. It is up to the user to ensure that the function works properly.

```
int mcb_setTransitionRatioFunction(double (*ratio)(const glb_params,
const glb_params, const glb_params, const int*, const int*, int,
int, double, void*), void* udata)
```

If **mcb_setStepProposalFunction** is used to set the proposal function to an asymmetric function (*i.e.*, $\mathcal{W}(\hat{x} \rightarrow x_n) \neq \mathcal{W}(x_n \rightarrow \hat{x})$), then this will affect the acceptance criteria that should be used in the Metropolis–Hastings algorithm. This method is used to implement this change of acceptance criteria and sets the function that is used to compute the ratio $\mathcal{W}(\hat{x} \rightarrow x_n)/\mathcal{W}(x_n \rightarrow \hat{x})$. If this method is not called, or **ratio** is **NULL**, then it will be assumed that the ratio is 1. The function **ratio** should take nine (9) arguments according to the following specification (see App. A.1 for the Monte Carlo theory notation):

1. **const glb_params**: The current step x_n in the simulation.
2. **const glb_params**: The proposed new step $\hat{x}$ in the simulation.
3. **const glb_params**: The step sizes used in the simulation.
4. **const int***: An integer array with flags notifying if a parameter should be free in the simulation or not (1 for free, 0 for fixed).
5. **const int***: An integer array with flags notifying if a density should be free in the simulation or not (1 for free, 0 for fixed).
6. **int**: The number of free parameters **Nf** in the simulation (*i.e.*, the number of ones in argument 4).
7. **int**: The number of free densities **Nd** in the simulation (*i.e.*, the number of ones in argument 5).
8. **double**: A parameter denoting the relative step size. This will be one in the simulation, but the starting samples in the simulation will be determined from taking a step with a larger step size.

9. **void***: This argument can contain user defined data. The **udata** passed to **mcb_setTransitionRatioFunction** is passed in this argument.

The parameters passed to this function will essentially be the same as those passed to the step proposal function (except the proposed new step, which is the output of the step proposal function, and the random number array, which is not needed).

ratio – The function to use to compute $\mathcal{W}(\hat{x} \rightarrow x_n)/\mathcal{W}(x_n \rightarrow \hat{x})$. If **NULL**, the ratio will be set to 1.

udata – User defined data to be passed to **ratio**.

Returns:

MCB_OK as it will always be possible to set the function and user data pointers. It is up to the user to ensure that the function works properly.

```
int mcb_setExplicitRuleRates (int exper, int rule, char* file)
```

Using this method, it is possible to set the rule rates of any experiment and rule to user defined values. The rule rates are then used by **GLOBES** to compute the χ^2 function. The main usage of this method is to input real or simulated results from an experiment and compute the χ^2 using these instead of the ones computed by **GLOBES**. In order for this method to work properly, it must be called after **glbSetRates**. If this is not the case, the memory storage for the rule rates will not have been allocated by **GLOBES**. However, what input values that are used for the **glbSetRates** does not matter when computing the χ^2 if the rule rates have been set using this method.

exper – The index of the experiment for which to set the rule rates.

rule – The rule for which to set the rates.

file – A file containing the rates to set. The file should contain rates separated by whitespace characters or linebreaks (the method uses **fscanf** with the pattern "%f" to scan the file for each bin). The file must contain as many rates as the **exper** has bins.

Returns:

MCB_OK if the file was readable. **MCB_IO_ERR** if it was not.

```
int mcb_setSimulatedRuleRates (int exper, const glb_params in)
```

Using this method, it is possible to set the rule rates of any experiment to Poisson distributed random numbers with mean values given by the prediction of oscillation parameters `in`. The rule rates are then used by `GLoBES` to compute the χ^2 function. The main usage of this method is to simulate results from an experiment and compute the χ^2 using these instead of the ones computed by `GLoBES`. Thus, this can be used to perform an actual Monte Carlo determination of the confidence belts, rather than relying on the test statistic actually taking a χ^2 distribution. Only the rates for `exper` will be changed.

`exper` – The index of the experiment for which to set the rule rates. Use `GLB_ALL` to set the rule rates for all experiments.

`in` – The parameters to be used to predict the mean values of the Poisson distributions used.

Returns:

`MCB_OK` if the file was readable. `MCB_IO_ERR` if it was not.

```
int mcb_addOutDataFunction (double(*fcn)(glb_params), const char*
    fcnName)
```

This method can be used to add an arbitrary function of the oscillation parameters to the output files. From the perspective of the output files, the value of the function is treated as an extra parameter and is added after all other parameters.

`fcn` – The function of the oscillation parameters to add to the output files.

`fcnName` – The name of the function to use in the summary file.

Returns: `MCB_OK` if the function was added. `MCB_SET_ERR` if too many out data functions have already been added, or `MCB_ALLOC_ERR` if memory to store the function could not be allocated.

```
int mcb_clearOutDataFunctions ()
```

This method clears all of the out data functions set by `mcb_addOutDataFunction`.

Returns:

`MCB_OK` as the clearing will always be possible.

```

Nchain      // Number of chains
Npar        // Number of parameters
r           // Convergence criteria
free        // Flags for free parameters
Nburn       // Length of burn-in
Nmin        // Minimum chain length used
Nmax        // Maximum chain length used
time        // Time elapsed during simulation
Nsamp       // Number of samples produced
start_values // Starting values for the parameters
step_size   // Typical step size used in simulation
Npoint      // Number of distinct samples
varNames    // Variable names
T           // Temperature used in simulation

```

Table 2.1: The structure of the summary files (// and text thereafter is not written in the files). Each line except `Npoint` and `varNames` (see text) represents a new line in the summary file.

2.3 Definitions of outfiles

An important part of the `MonteCUBES` interface is the creation of files storing the samplings produced by the MCMC, and we will refer to them as the *raw sampling files*. This is the output of the `MonteCUBES` C library and the input of the `MonteCUBES` Matlab GUI. If you are planning to use both of these, you can most probably skip this section, since the Matlab GUI will take care of reading and interpreting the raw sampling files for you. However, if you are planning to create your own method of plotting the results, or if you want to search for irregularities within the samplings, this section describes the structure of the raw sampling files.

There are two types of raw sampling files, the *summary files* and the *chain files*. Each of these are described in subsections below.

2.3.1 Summary files

For each MCMC simulation, a summary file will be created. This file contains general information about the MCMC simulation, such as which parameters that were used, the duration of the simulation, how many samples were produced, etc. The summary file will be named `<name>.mcb`, where `<name>` is the `outfile` argument passed to the MCMC (see Sec. 2.2).

In Tab. 2.1, we give the structure of the summary files. The entries of this table should be interpreted in the following way:

Nchain – The number of chains produced in the simulation.

Npar – The number of parameters used in the simulation.

- r** – The convergence criteria put on the parameters. This consists of **Npar** numbers, one for each parameter.
- free** – Flags denoting which parameters that were allowed to vary during the simulation. This consists of **Npar** numbers, where 0 denotes that the parameter was fixed and 1 that it was allowed to vary.
- Nburn** – The number of samples per chain that were not considered in the convergence criteria.
- Nmin** – The minimum number of samples per chain that the MCMC was told to produce after burn-in.
- Nmax** – The maximum number of samples per chain that the MCMC was told to produce after burn-in.
- time** – The running time (in seconds) of the MCMC simulation.
- Nsamp** – The actual number of samples produced per chain by the MCMC simulation after burn-in.
- start_values** – The typical start value of the chains. This line contains **Npar** numbers, one for each parameter. The actual starting values for the chains are chosen by taking a random step away from these values (this step is three times larger than the random steps taken during the MCMC). If more than one start position have been used, then this is the first of the start positions to be stored in memory.
- step_size** – The typical step sizes for the different parameters. This line contains **Npar** numbers, one for each parameter.
- Npoint** – This actually represents **Nchain** different lines. The lines contain the number of distinct samples in the different chains (starting with chain 1). This number includes the burn-in samples and should equal the number of lines in the corresponding chain files.
- varNames** – This represents **Npar** different lines. Each line contains the name of one of the variables used in the simulations within quotation marks. The MonteCUBES Matlab GUI will display these variable names.
- T** – This is a single decimal number containing the temperature used in the simulation. This must be known in order to cool the chain properly.

2.3.2 Chain files

The chain files contain the actual samples created in the MCMC simulations. Unlike the summary files, several chain files will be produced by the same MCMC run (unless only one chain is used). The naming convention for the summary files is `<name>.mc<n>`, where `<name>` is the `outfile` argument passed to the MCMC (see Sec. 2.2) and `<n>` is the chain number. Thus, a MonteCUBES

simulation using three chains and specifying `out` as the filename would produce the files `out.mcb`, `out.mc1`, `out.mc2`, and `out.mc3`. Since the summary files and chain files have the same names (up to the suffixes), it is easy to tell which chain files have been created from reading the summary file.

The chain files contain all of the samples produced in the MCMC simulation, including the burn-in samples. However, in order to make the file size smaller, the chain files only store each sample once along with a number indicating how many times that point was sampled before leaving. Each line of a chain file has the following structure:

`chi2 N par`

The interpretation of this is:

`chi2` – The value of the χ^2 function for this sample.

`N` – The number of times this point was sampled by the MCMC.

`par` – A list of `Npar` (see the summary file description) numbers representing the different parameter values.

2.4 Examples

2.4.1 Simulation of the ISS neutrino factory

Table 2.2 contains a typical example program designed to simulate the ISS neutrino factory [4]. Do not worry if you find the font too small, we will go through each line of code separately. This example is essentially how you would construct a `MonteCUBES` simulation without caring too much about implementing new physics, priors, or degeneracies.

Let us start from the beginning of the code with the `#include` statements:

```
#include <math.h>
#include <globes/globes.h>
#include <montecubes/montecubes.h>
```

As you will notice, there is essentially nothing strange in these lines. We need `math.h` to tell us about `M_PI`, `globes.h` to have access to the `GLoBES` functions, and `montecubes.h` in order to use the functions introduced by `MonteCUBES`.

If you are unfamiliar with C programming, the `main` function is the function that is called when you run the binary executable file that is constructed by the compiler. Thus, we will not explain its syntax in any detail as this can be found in any tutorial on C programming. Instead, we simply focus on the instructions within, which are executed when the binary is. The first statements, *i.e.*,

```
double theta12 = 33.21*M_PI/180;
double theta13 = 0*M_PI/180;
double theta23 = 45*M_PI/180;
double deltacp = M_PI/2;
```

```

#include <math.h>
#include <globes/globes.h>
#include <montecubes/montecubes.h>

int main(int argc, char *argv[]){
    double theta12 = 33.21*M_PI/180;
    double theta13 = 0*M_PI/180;
    double theta23 = 45*M_PI/180;
    double deltacp = M_PI/2;
    double sdm = 8.0e-5;
    double ldm = 2.5e-3;
    double r = 0.025;

    glbInit(argv[0]);
    glbInitExperiment("ids-baseline.glb",&glb_experiment_list[0],&glb_num_of_exps);

    glb_params true_values = glbAllocParams();
    glb_params start_values = glbAllocParams();
    glb_params steps = glbAllocParams();
    glb_params convcrit = glbAllocParams();
    glb_params input_errors = glbAllocParams();
    glb_projection pro = glbAllocProjection();

    glbDefineParams(true_values,theta12,theta13,theta23,deltacp,sdm,ldm);
    glbSetDensityParams(true_values,1.0,GLB_ALL);
    glbSetOscillationParameters(true_values);
    glbSetRates();
    glbSetCentralValues(true_values);

    glbCopyParams(true_values,start_values);

    glbDefineParams(input_errors,0.04*theta12,0,0.1*theta23,0,0.04*sdm,0.1*ldm);
    glbSetDensityParams(input_errors,0.02,GLB_ALL);
    glbSetInputErrors(input_errors);

    glbDefineProjection(pro,GLB_FREE,GLB_FREE,GLB_FREE,GLB_FREE,GLB_FREE,GLB_FREE);
    glbSetDensityProjectionFlag(pro,GLB_FREE,GLB_ALL);
    glbSetProjection(pro);

    int m;
    for(m = 0; m < glbGetNumOfOscParams(); m++){
        glbSetOscParams(convcrit,r,m);
        glbSetDensityParams(convcrit,1.0,GLB_ALL);
        mcb_setConvergenceCriteria(convcrit);

        glbCopyParams(true_values,steps);
        glbDefineParams(steps,0.005*theta12,0.0025,0.005*theta23,0.2*M_PI,0.005*sdm,0.005*ldm);
        glbSetDensityParams(steps,0.02,GLB_ALL);

        mcb_setBurnNo(MCB_DYNAMIC_BURN);
        mcb_setLengthMax(10000000);
        mcb_setLengthMin(2000);
        mcb_setConvergenceCheck(10000);
        mcb_setStepSizes(steps);
        mcb_addStartPosition(start_values);

        mcb_MCMC("idsres",GLB_ALL,GLB_ALL);

        glbFreeParams(true_values);
        glbFreeParams(convcrit);
        glbFreeParams(start_values);
        glbFreeParams(input_errors);
        glbFreeParams(steps);
        glbFreeProjection(pro);
    }
}

```

Table 2.2: MonteCUBES C source code for simulating the ISS neutrino factory.

```
double sdm = 8.0e-5;
double ldm = 2.5e-3;
double r = 0.025;
```

are simply declarations of variables that we will use later on in our program. You will recognize the first six values as neutrino oscillation parameters, with θ_{13} set to zero and the other parameters according to the ISS neutrino factory simulation definitions. The last parameter, `r`, is a parameter that we will use to set the convergence criteria for our Markov Chains. A smaller value would imply more stringent convergence criteria (this value is already quite stringent).

We now move on to initializing GLoBES and the experiments we want to use. This is done through the statements:

```
glbInit(argv[0]);
glbInitExperiment("ids-baseline.glb",
                  &glb_experiment_list[0], &glb_num_of_exps);
```

The first of these statements initializes GLoBES (see the GLoBES manual for details). The second tells GLoBES that we want to use `ids-baseline.glb` as the AEDL file, which contains a description of the ISS neutrino factory setup.

Once GLoBES has been initialized, we need to define the parameter vectors we are going to use, as well as allocate memory where the parameter vectors can be stored. We also need to define a projection which will tell MonteCUBES what parameters it should be changing in the Markov Chains. Out of the six the statements

```
glb_params true_values = glbAllocParams();
glb_params start_values = glbAllocParams();
glb_params steps = glbAllocParams();
glb_params convcrit = glbAllocParams();
glb_params input_errors = glbAllocParams();
glb_projection pro = glbAllocProjection();
```

the first five declare and allocate memory to parameter vectors, while the last statement declares and allocates memory to a projection. The five parameter vectors we will use are:

- `true_values` – We will use these values to set the event rates. They correspond to the values we assume have been realized in Nature.
- `start_values` – These are parameter values at which we will start the Markov Chains. The first sample in the Markov Chains will be given by a rather large step (three times the normal step size) away from this value.
- `steps` – We will use this parameter vector to store the typical step sizes that we want to use in the Markov Chains. The steps will be Gaussian with these values as standard deviation.

- `convcrit` – This parameter vector stores the convergence criteria. Thus, we can in principle set different convergence criteria for different parameters, although we will not do this in this example.
- `input_errors` – As in standard GLoBES programs, we will need to set the input errors that should be used by the prior functions. This parameter vector will be used to store these errors.

It is now time to set the event rates that would be expected given that the neutrino oscillation parameters take the values we declared in the beginning of the `main` function. This is done through the statements:

```
glbDefineParams(true_values, theta12, theta13,
                theta23, deltacp, sdm, ldm);
glbSetDensityParams(true_values, 1.0, GLB_ALL);
glbSetOscillationParameters(true_values);
glbSetRates();
glbSetCentralValues(true_values);
```

The first line sets the values stored in `true_values` to the neutrino oscillation parameters from the beginning of the `main` function. The second line tells GLoBES that the matter density along the baselines are supposed to be the same as those defined in the AEDL files used. In the third line, the neutrino oscillation parameters that should be used to compute rates is set to `true_values`. Finally, in the fourth line, we tell GLoBES to set the event rates of all experiments to the ones that would be expected if the neutrino oscillation parameters were those in `true_values`. Finally, we will also use the `true_values` as the central values when computing the priors, which is what is accomplished by the fifth line. We will also use these same values as the starting values of our Markov Chains, thus we simply copy the parameter vector by the statement:

```
glbCopyParams(true_values, start_values);
```

In order to compute the prior properly, we also need to define the input errors. The following three statements defines the input errors to the values specified in the ISS neutrino factory simulation and tells GLoBES to use these values to compute the prior:

```
glbDefineParams(input_errors, 0.04*theta12, 0.0.1*theta23,
                0, 0.04*sdm, 0.1*ldm);
glbSetDensityParams(input_errors, 0.02, GLB_ALL);
glbSetInputErrors(input_errors);
```

As mentioned earlier, we will also need to tell the Monte Carlo which parameters that should be allowed to vary in the simulation. To this end `MonteCUBES` will use the projection which is currently set in GLoBES. Thus, it can be set as:

```
glbDefineProjection(pro, GLB_FREE, GLB_FREE, GLB_FREE,
                  GLB_FREE, GLB_FREE, GLB_FREE);
glbSetDensityProjectionFlag(pro, GLB_FREE, GLB_ALL);
glbSetProjection(pro);
```

The first line sets all neutrino oscillation parameters free in the simulation, while the second line does the same for the matter densities. Finally, the last line tells GLoBES to use this projection.

It is now time to start worrying about what we put into the Monte Carlo simulation. An important part of this is telling it how well the chains should have converged before the Monte Carlo is terminated. This is done through the following code:

```
int m;
for(m = 0; m < glbGetNumOfOscParams(); m++)
    glbSetOscParams(convcrit,r,m);
mcb_setConvergenceCriteria(convcrit);
```

To start with, `m` is simply declared, since we will need it as a loop variable. The `for` loop simply sets the convergence criteria for all oscillation parameters to `r`, which was defined in the beginning of the program. Note that we could just as well have used the `glbDefineParams` here as well. However, this `for` loop would be sufficient also if we had used a user defined probability engine with more parameters (see the GLoBES manual). Finally, the last statement tells MonteCUBES to use these values as the convergence criteria.

Another just as important thing to tell the Monte Carlo is how big steps it should be taking. To this end, we have defined the `steps` parameter vector, which will be passed to the function starting the simulation. We simply need to put the typical step length into the vector using standard GLoBES commands:

```
glbDefineParams(steps,0.005*theta12,0.0025,
               0.005*theta23,0.2*M_PI,0.005*sdm,0.005*ldm);
glbSetDensityParams(steps,0.02,GLB_ALL);
```

The typical step length that should be used is about the one standard deviation that we expect. If the step length is too small, the Monte Carlo will take more time to converge. If it is too large, then the sampling of the parameter space will be bad as the chains will get stuck in the same point for a long time.

It is now time to set options for the Monte Carlo that we do not wish to be put to their defaults. Here, we use four examples:

```
mcb_setBurnNo(MCB_DYNAMIC_BURN);
mcb_setLengthMax(10000000);
mcb_setLengthMin(2000);
mcb_setStepSizes(steps);
mcb_addStartPosition(start_values);
```

The first line tells the Monte Carlo to use dynamic burning (to read more about the burn-in process, see App. A). This means that the Monte Carlo will initially produce chains with twice the number of samples as the minimum chain length. If the last half of the chain does not fulfill the convergence criteria, then the Monte Carlo doubles the number of samples and use the earlier chain as burn-in. This continues until the last half of the chain has reached convergence or

the maximum chain length has been reached. The following two lines set the maximum and minimum length of the chains to 10^7 and 2000, respectively. An equivalent statement would be:

```
mcb_setLengthMinMax(2000,10000000);
```

Since we are using dynamic burning, the initial burn-in will be 2000 samples and another 2000 samples will be produced before checking convergence. The following line sets the typical step size to use in the Monte Carlo to the **steps** parameter vector, while the last line adds a nominal starting value for the chains. The chains will then actually be started from points which are one step away from this value (with a step size of three).

It is now time to start our Monte Carlo simulation. This is done by the statement:

```
mcb_MCMC("idsres",GLB_ALL,GLB_ALL);
```

The first argument is a string which constitutes the base for the output filenames (see the separate section on output files). The two final arguments will be passed to the χ^2 function of GLoBES and represent which of the defined experiments and rules that we wish to use. Since we want to use all of the experiments and rules defined in the AEDL files, we use **GLB_ALL**.

Finally, the Monte Carlo has been run and it is time to end the program. It is good practice to free the memory allocated in the program, even though the program will terminate immediately afterwards. Therefore, we end our program with:

```
glbFreeParams(true_values);
glbFreeParams(convcrit);
glbFreeParams(start_values);
glbFreeParams(input_errors);
glbFreeParams(steps);
glbFreeProjection(pro);
```

2.4.2 Treating degeneracies

An important aspect of future neutrino oscillation searches is the appearance of degeneracies. In general, if the degeneracies are well-separated and the starting point is close only to one of the degenerate solutions, then all chains will fall into that degenerate solution with a miniscule probability of ever jumping to another of the degenerate solutions unless simulations run for about the age of the Universe.

On the other hand, if the starting point is located in such a way that chains fall into different degeneracies, the low probability of changing minimum for a chain will result in chains with very bad convergence criteria.

In order to solve the problem with degeneracies, **MonteCUBES** includes a procedure for sampling all of the degeneracies in the same simulation and with

appropriate weights. This is implemented by the possibility of adding *degeneracy steps* to the simulation. Essentially, a degeneracy step is a constant step in parameter space describing the distance between two degeneracies. The inclusion of a degeneracy step can be viewed as digging a tunnel between the degeneracies so that the Monte Carlo chains do not have to cross the large barrier between them in order to sample them both. For details on how this works, see App. A. In this section, we will make an example of how to search for degeneracies and resolve them using the **MonteCUBES** C library. The example will be continued in the next chapter, where we will display the results using the **MonteCUBES** Matlab GUI. Appendix A also includes a description of a gradual temperature lowering algorithm to find the different degenerate solutions. Also this algorithm is implemented in **MonteCUBES** and, for illustration, we will use it to find the degeneracies automatically (even if we know where the degeneracies are located).

First of all, in order to need the degeneracy solver, we need an experimental setup which has degeneracies. In particular, in order to demonstrate the efficiency of the degeneracy resolution, we choose the following experimental setup with only the $L = 2000$ km detector from Ref. [5].

The above setup has a twofold degeneracy which shows up at different values for θ_{13} and δ and a sign change of Δm_{31}^2 . Since the two degeneracies are located in different mass hierarchies, there is a huge barrier between them and the chains have to pass $\Delta m_{31}^2 = 0$ (essentially the no-oscillation region) in order to jump between the degeneracies. Additionally, the degenerate solution is not a perfect fit and it should be sampled slightly less often than the true solution. Thus, testing this setup will also show that the degeneracy step solution is able to reproduce the different solutions with the appropriate weights.

In principle, the degeneracy finder is a Markov Chain Monte Carlo by itself. Thus, in order to run it, the user must specify the same parameters before running it as if running the main method of **MonteCUBES** (such as step sizes). The degeneracy finder is then run by a command similar to:

```
mcb_TdegFinder(1000.0, 0.01, 10, 20,
               GLB_ALL, GLB_ALL, 20.0, 1, "degLocs.mcd");
```

Here, the chains are started at a temperature of $T_h = 1000$ and gradually cooled to $T_l = 0.01$, the number of temperature steps is ten and the number of chains tried is 20. The `GLB_ALL` arguments tells the degeneracy finder to use all experiments and rules, the 20.0 is the maximum value of the χ^2 which should be considered a degenerate solution, the 1 tells the degeneracy finder to automatically set the degeneracy steps using the function `mcb_setDegeneracyStep`, and the final string `"degLocs.mcd"` is the name of the file into which the degenerate solutions are stored.

Since it can take some time to run this method, it may be advisable to run it only once (assuming that the experimental setup does not change), and instead use

```
mcb_readDegeneracySteps("degLocs.mcd");
```

in subsequent runs of the program. This will simply load the degeneracies stored in `degLocs.mcd` and set the correct degeneracy steps.

Once the appropriate degeneracy steps have been set, the usual Markov Chain Monte Carlo can be run as described in the previous example.

Chapter 3

The MonteCUBES Matlab GUI

In this chapter, we will describe the MonteCUBES Matlab GUI and how to use it. In order to do so, we will use the results from the examples in the previous chapter. The GUI will be presented on an example basis in which we use it to produce the different plots and explain the features as we go along.

3.1 Starting the GUI and reading simulation results

In order to start the MonteCUBES Matlab GUI, you should first start Matlab and make sure that the GUI code is located in the current directory (the GUI code is distributed in the sub-directory `matlab` of the MonteCUBES distribution). Once this is the case, the GUI can be started by simply typing `MonteCUBES` in the command window.



The GUI will then start up in a new window in a state reminiscent of that in Fig. 3.1. The startup appearance of the GUI show only the most basic features, simply because no simulation results have been read, which naturally means they cannot be plotted. The main features of the window are the following:

- The *Open .mcb* button. Since you just started the GUI, this is probably the feature you want to use. Pressing the button will provide you with a file dialog asking you which `.mcb` file you wish to open. The read process will be described below. This button will always be visible in the GUI.
- The *Exit* button. This button is fairly self-explanatory. Pressing it will simply close the GUI window. It will always be visible in the GUI.

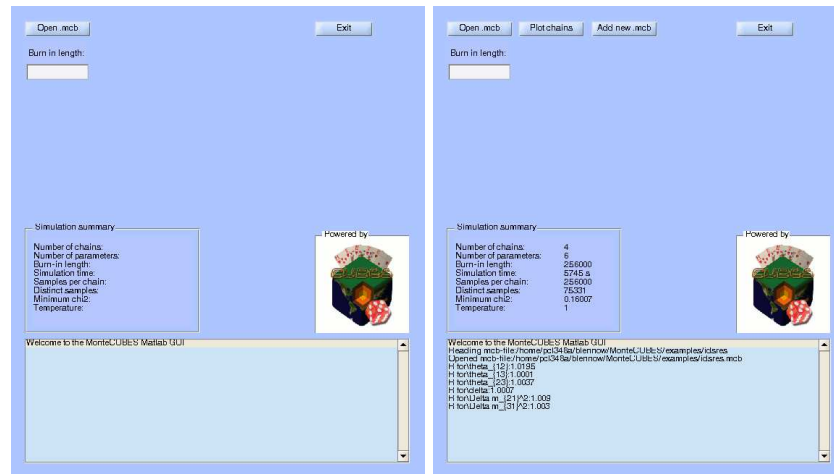


Figure 3.1: The MonteCUBES Matlab GUI at startup and after reading simulation results.

- The *Burn length* field. This field is specific to the startup appearance of the GUI. By default, the GUI will get the information on the burn length from the summary file. However, if the user inputs a number into this field, it will be used instead of the pre-defined burn-length.
- The *Simulation summary*. Since no simulation files have been opened, this part currently displays no information. However, when a simulation has been read, it will provide general information about the simulation and its results. This part of the GUI is always visible.
- The *Feedback* area. The feedback area should currently display a message saying “Welcome to the MonteCUBES GUI”. Additional messages will appear in the feedback area as the GUI is used. The feedback area is always visible.

Once the GUI has been started, it must be provided with the output from the simulation of which it should produce high-level information, such as contour plots. As just hinted above, this is done by pressing the *Open .mcb* button. The user will then be provided with a file dialog in order to select the summary file of the simulation. The GUI will then start by reading the summary file in order to get the basic information about the simulation as well as to deduce what chain files to read. It is assumed that the chain files (see Sec. 2.3) are located in the same directory as the summary file.

If the *Burn length* field is left blank, then the GUI will read the simulation files assuming that the burn length is the same as the one provided in the summary file. However, if the burn length field evaluates to a positive integer, this will override the burn length in the summary file. Regardless of which, the chain

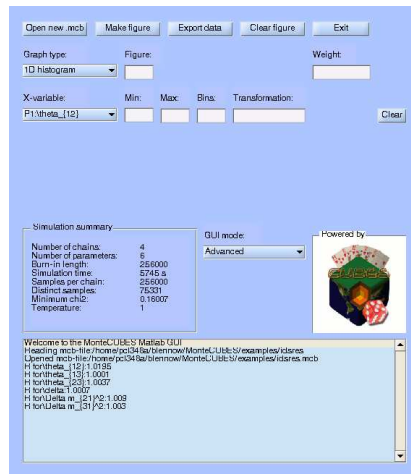


Figure 3.2: The appearance of the MonteCUBES Matlab GUI after pressing the plot chains button.

files will be read into the GUI with the given burn length and information on the convergence of the chains will be written to the feedback area. In addition, the information from the summary file is also stored by the GUI and the simulation summary provides basic information on the currently loaded simulation.

Once a simulation has been loaded, the *Plot chains* button will appear. If the user is satisfied with the simulation that has been loaded, pressing this button will make the GUI proceed to plotting mode, where a number of different plots can be produced from the loaded data. If the user wants to load a different file, or use a different burn length, this can be done by pressing the *Open .mcb* button again. In addition, the *Add new .mcb* button provides the user with the possibility of joining the chains from different simulations. This will append the results of the second simulation with those from the first. Several checks will then be made (such as checking that the same parameters are free in both simulations) and the user will be notified in the feedback window if anything is strange.

In Fig. 3.1, and for most of this manual, we are using the simulation files resulting from running the program presented in Sec. 2.4.1. In the end of this chapter, we will switch to the results from Sec. 2.4.2.

3.2 Making plots

After deciding to plot the loaded simulation by pressing the plot chains button, the GUI will change in appearance to look like Fig. 3.2. The GUI now contains several new elements:

- The *GUI mode* selector. This sets the graphical mode of the GUI and can be set to *Simple* or *Advanced* (the default). If set to *Simple*, a lot of the GUI controllers will be hidden and default values used (except for the filter parameter, which will be set to one). The simple mode can be useful for the user who only wants to plot simulation results with as little effort as possible, although we recommend to use the advanced setting in order to take full advantage of the **MonteCUBES** capabilities.
- The *Make figure* button. Pressing this button will produce a new figure according to the current state of the GUI.
- The *Export data* button. This button will save the data necessary to produce a plot according to the current state of the GUI. The user will be provided with a file dialog in order to choose where to save the data. The structure of the output file depends on the type of graph that should be produced. This option is not available for the triangle and 3D surface plots, see below.
- The *Clear figure* button. Fairly self explanatory, this button will clear the figure currently indicated in the *Figure* parameter field (see below). If the Figure parameter field is empty, this button has no effect.
- The *Graph type* selector. This drop-down menu provides the user with a number of different plot types which can be produced by the GUI. The visual appearance of the GUI will depend on the graph type that is actually selected.
- One or more *parameter fields*. Just right of the graph type selector, a number of parameter fields will appear depending on the selected graph type. For example, the *Figure* field will always be present and is used to tell the GUI in which figure to plot the results when pressing the make figure button. If the figure field is empty, then the results will be plotted in a new figure. Other parameter fields may be specific to each graph type and will be explained along with the graph types.
- One or more *variable rows*. Depending on the dimensionality of the selected graph type, a number of variable rows will appear (*i.e.*, a plot requiring one variable displays one variable row and so on). Each variable row contains the following elements, which are only visible in the cases where they affect the resulting plot:
 - *Variable*: The drop-down menu to the left can be used to pick what variable to use for this dimension of the plot. It contains all of the variable names from the simulation summary file, as well as the variable number used to represent the parameter in the transformation field (see below).
 - *Min*: The minimum variable value to use when a plot is produced. If left blank, the default value is the minimum value of the variable in the simulation.

- *Max*: The maximum variable value to use when a plot is produced. If left blank, the default value is the maximum value of the variable in the simulation.
- *Bins*: The number of bins in which to divide this variable if the graph type is such that binning is needed. If left blank, this defaults to 30.
- *Transformation*: This is an arbitrary transformation of variables that may be applied to the simulated values. A transformation should be written in such a way that it is a `Matlab` function of variables `P1`, `P2`, `...`, each referring to one of the simulation parameters. Furthermore, it should be written in such a way that its result is a vector if the variables are vectors (*i.e.*, if the transformation should be such that `P1` and `P2` are multiplied, the transformation should be `P1.*P2` rather than `P1*P2`). This allows the user to use any combination of parameters as the variables of the plots. The variable resulting from this transformation is the variable that will actually be used in the plots. If this field is left blank, then the variable chosen in the variable drop-down menu will be used. See below for examples.

Note! Unless the full transformation used has a jacobian of one, this will effectively change the prior used in the simulation. In the graph types where this matters, it can be counteracted by the use of a weight function.

- *Boundary cond.*: This is a drop menu which is used in order to set the boundary conditions of the filtering functions. There are four different options:
 1. *Zero padded*. With this boundary condition set, the GUI assumes that any bins outside of the original grid of bins contain zero samples.
 2. *Constant continuation*. This boundary condition means that the GUI will assume that the last bins in the original grid are repeated infinitely (*i.e.*, for the whole reach of the filter).
 3. *Cyclic*. With a cyclic boundary, the GUI assumes that the continuation of a boundary is given by the bins on the opposite boundary. This is useful for complex phases and other cyclic parameters which are plotted within a full period.
 4. *Mirrored*. The mirroring boundary condition makes the GUI assume that the bins are mirrored in the boundary.

This only affects the plots where filters are applied. If a two- or three-dimensional graph type is chosen, there will also be *Variable switch* buttons between the variable rows. Pressing such a button will exchange the values in the two variable rows adjacent to it.

- The *Clear* button. Pressing this button will clear all of the text fields in the variable row.

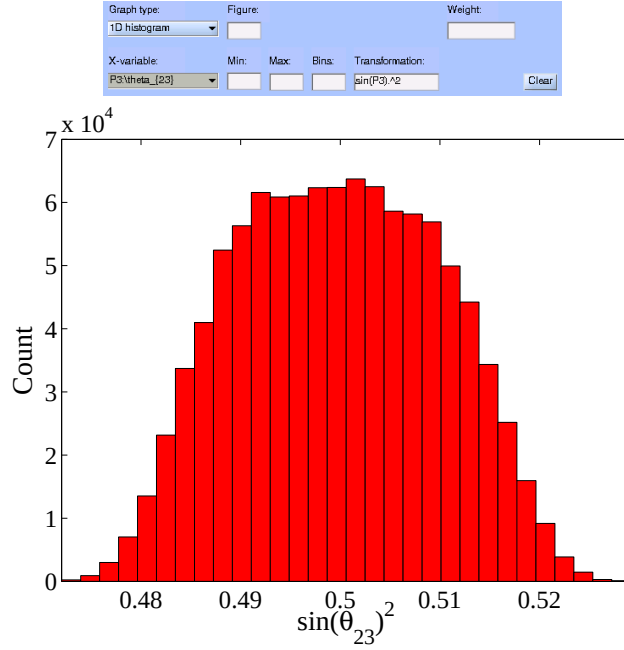


Figure 3.3: A 1D histogram plot for $\sin^2(\theta_{23})$ for the ISS neutrino factory simulation, as well as the GUI settings that produced it. Note that we have not used a weight factor. The reason for this is that, although the transformation used is non-linear, it is almost linear in the region of interest.

3.2.1 1D histogram plots

The one-dimensional histogram plots simply divides the plotting interval into bins, counts the number of samples in each bin, and uses the result to produce a histogram. An example of this is shown in Fig. 3.3. The figure also shows the state of the GUI that was used to produce it. Note that the transformation of θ_{23} to $\sin^2(\theta_{23})$ includes `.^` rather than `^` in order for the squaring of the vector `sin(P3)` to be done element by element. In this example, we have chosen to use 20 bins rather than the default value of 30 simply to demonstrate the usage of the binning parameter. Note that the GUI has also parsed the variable transformation string into the proper x -label of the figure. This will be done in every graph type. There is one additional parameter field affecting the histogram plots, this is the *weight* function parameter field. It provides the user with the possibility of weighting the number of samples depending on the parameters. The number of times a point has been sampled is simply multiplied with this weight. The value put into this field is parsed into an inline function, similar to the transformation of variables in the variable rows. Just as with the variable transformations, the user must make sure that the function computes

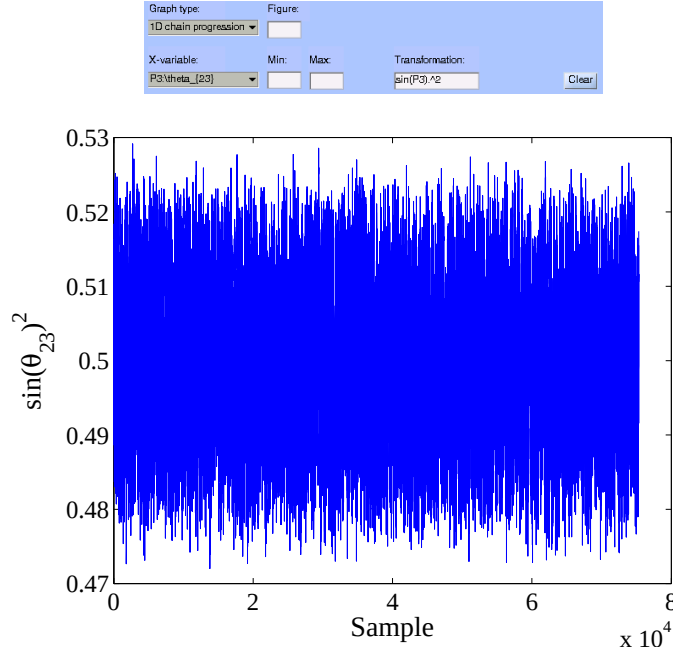


Figure 3.4: A 1D chain progression plot for $\sin^2(\theta_{23})$ for the ISS neutrino factory simulation, as well as the GUI settings that produced it.

the weights element-wise as it is applied to the full set of samples at the same time. If no weight function is specified, then the original number of samples is used. This feature is useful to counteract the effects of a variable transformation with a jacobian different from one.

Using the export data feature with a 1D histogram graph type produces a file containing two columns of floating point numbers. The first of these represents the variable values at the center of the bins, while the second represents the number of counts.

3.2.2 1D chain progression plots

In Fig. 3.4, we show a one-dimensional chain progression plot for $\sin^2(\theta_{23})$. This graph type simply provides a plot of the variable value against the number of the distinct sample. It is a tool for visualizing how the chains behave. A well-behaved and well-converged chain should seem to jump randomly back and forth within the range of values taken by the chain. In addition, it should not be possible to distinguish where one chain ends and another one starts. A chain which actually looks as a random walk within the parameter space usually has bad convergence and should not be trusted. Two possibilities of fixing this problem are using larger steps or producing more samples (or a combination of

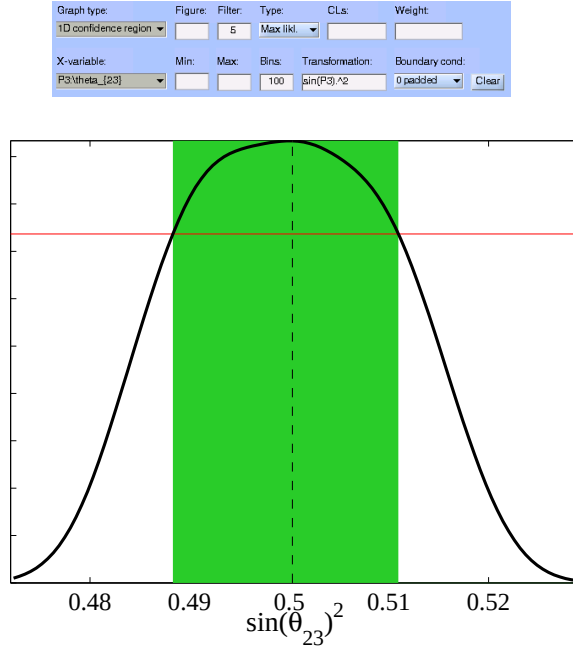


Figure 3.5: A 1D confidence region plot for $\sin^2(\theta_{23})$ for the ISS neutrino factory simulation, as well as the GUI settings that produced it.

the two).

Using the export data feature with a 1D histogram graph type produces a file containing two columns of numbers. The first of these is simply the index of the distinct sample in the chain (which increases by one per row), while the second represents the variable value at that sample.

3.2.3 1D confidence region plots

The one-dimensional confidence region plot is essentially an advance variant of the histogram plot. An example of this plot is shown in Fig. 3.5. The procedure followed by the GUI in order to produce the confidence region plot is as follows. First of all, the same information as for the histogram plot is produced, *i.e.*, the samples are divided into bins. The GUI then applies a Gaussian filter to the histogram in order to provide a smoother curve and get rid of the statistical fluctuations. The standard deviation of the Gaussian filter (in bins) is set by the *filter* parameter field. If the field is left blank, then the width is assumed to be zero and no filtering is applied. It is possible to use a non-integer standard deviation. Finally, the GUI computes what bins that are needed to contain a given fraction of the samples (using as few bins as possible). The fraction is set by the *confidence levels* (CLs) parameter field and the default value is 0.68. Just

as in the case of a 1D histogram plot, the confidence region plots are affected by the *weight* function parameter field.

The actual graph contains a thick black curve, representing the distribution of samples along the chosen variable, a horizontal red line, representing the level above which bins are needed to contain the given ratio of the samples, a vertical dashed black line, representing the best-fit bin, and a green region, representing the parameter values within which the ratio is contained. The graph has no scale on the *y*-axis, simply due to the fact that it represents a distribution and the normalization is arbitrary. The user is also presented with the option of producing one-sided confidence regions through the *Type* parameter field.

In the edges of the graph, the Gaussian filter assumes the boundary conditions set by the *Boundary cond.* variable field.

Using the export data produces a file containing the one-dimensional distribution. The main advantage over the export results from the histogram plot is that the data can be filtered.

3.2.4 2D scatter plots

The two-dimensional scatter plots simply puts points in a two-dimensional plot where the samples are located for the variables chosen. An example of a two-dimensional scatter plot is given in Fig. 3.6. Apart from providing a first view of how the samples are distributed in the chosen variables, the scatter plots may also provide a visual test for how well-behaved the simulation was. For a well-converged simulation, the points should be distributed quite smoothly and no trace of them being the result of a random walk should be visible (as in the figure). Chains that do not have a good convergence will display features as some regions being overpopulated due to the walk staying there for too long. Again, the remedy for resolving this issue would be to increase the number of samples in the simulation and/or changing the step sizes.

The use of the export data feature with this plot type will result in a list of the different points in parameter space. The main advantages of such a file over the original chain files is that the chains are joined, that the file only contains the information on the specific parameters, and that the parameters can be arbitrarily transformed (although any reasonable plotting tool should be able to handle this).

3.2.5 2D counts contour plots

Much like the scatter plots are the two-dimensional equivalents of the chain progression plots, the two-dimensional contour plots are the equivalents of the confidence region plots. An example of a contour plot is given in Fig. 3.7. The plots are constructed by binning the samples, using a Gaussian filter to eliminate the statistical fluctuations, and finally the smallest-area contours containing a given ratio of the samples are drawn.

Similar to the confidence region plots, the counts contour plots have two extra parameter fields, the *filter* field and the *confidence levels* (CLs) field. The

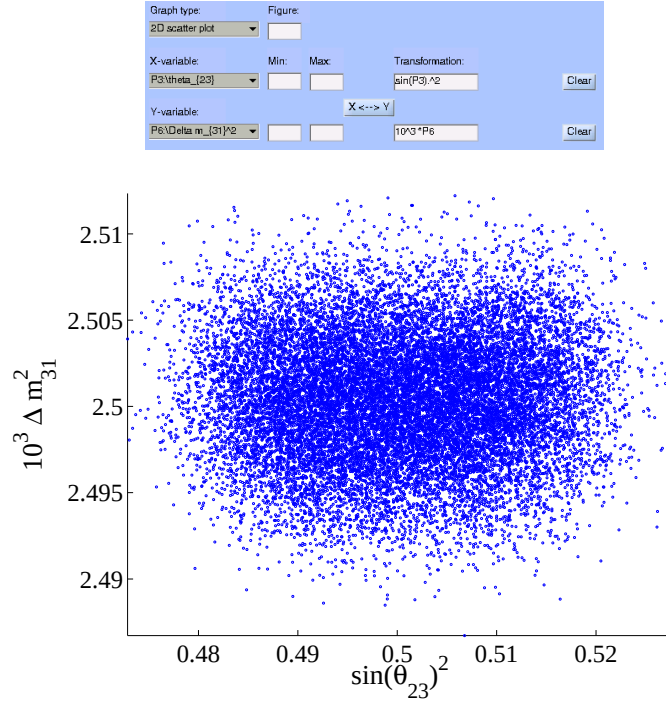


Figure 3.6: A 2D scatter plot for $\sin^2(\theta_{23})$ and Δm_{31}^2 for the ISS neutrino factory simulation, as well as the GUI settings that produced it. Note that this scatter plot only contains the last 64000 samples to reduce the figure size.

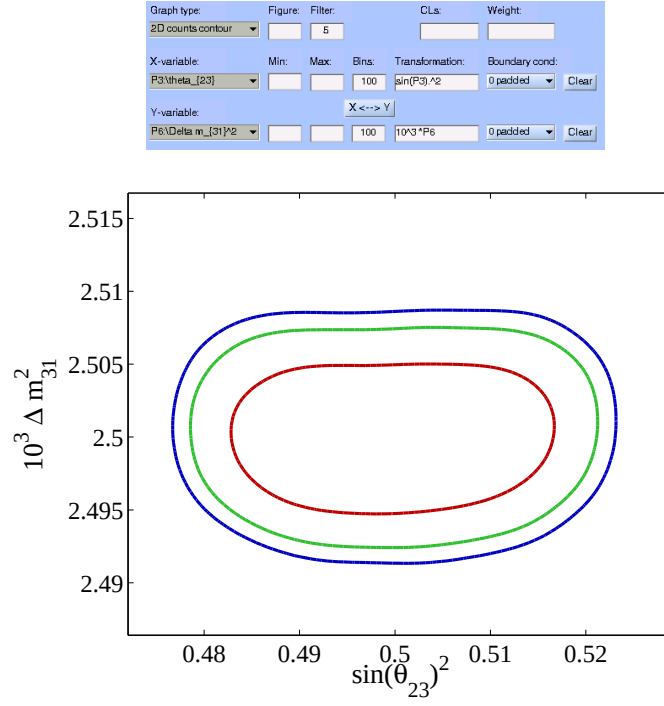


Figure 3.7: A 2D count contour plot for $\sin^2(\theta_{23})$ and Δm^2_{31} for the ISS neutrino factory simulation, as well as the GUI settings that produced it.

filter field can be used to input the standard deviation of the Gaussian filter to be used. The user can choose to use two different numbers in the x and y directions. This is done by simply inputting two different comma and/or white-space separated numbers in the filter field. The first of these will be used in the x direction and the second in the y direction. If the filter field is left blank, then no filter will be applied.

The CLs field can be used to input the levels at which to draw the contours. In order to draw contours at levels other than the pre-defined values, simply input a comma and/or white-space separated list of these into the field. The CLs are given in ratios compared to the total number of samples and should therefore be numbers between 0 and 1. If the field is left blank, then a default list of 0.68, 0.90, and 0.95 will be used.

Similar to the 1D histogram and confidence region plots, this graph type is also affected by the *weight* parameter field.

If using the export data feature with this graph type, the resulting file will contain a matrix defined by the `Matlab` contour definition.

3.2.6 Triangle plots

The triangle plot graph type can be used to produce a figure containing several of the confidence level and contour plots. A typical triangle plot is presented in Fig. 3.8. By default, the triangle plot selects the combination of three simulation variables that present the largest cumulative relative covariance and produces all of the confidence level and contour plots for these. This can be changed by filling in the *parameters* extra parameter field in order to select what parameters to use. When doing so, simply input a comma and/or white-space separated list with the numbers of the parameters to use (without the initial P).

Although two variable rows are displayed when the triangle plot type is selected, only the bins fields have any effect on the final result. This is simply due to the fact that the x and y parameters change throughout the triangle plot. The bins fields will however be used - the x bins field will be used in all plots, while the y bins field will only be used in the contour plots. Furthermore, the *filter* and *CLs* fields are also available for the triangle plot. The values of these fields will simply be passed along to the functions drawing the confidence region and contour plots and therefore have exactly the same usage as for these. An important point is that only the first value will be used in the case of the confidence region plots, since this plot type requires that only one value is passed to it. This means that an input of 0.68,0.9 for the CLs field will produce the same contours, but different confidence region plots, compared to an input of 0.9,0.68.

Since the data for the triangle plots can be exported using the individual 1D confidence region and 2D counts contour plots, the export data feature is turned off for the triangle plots.

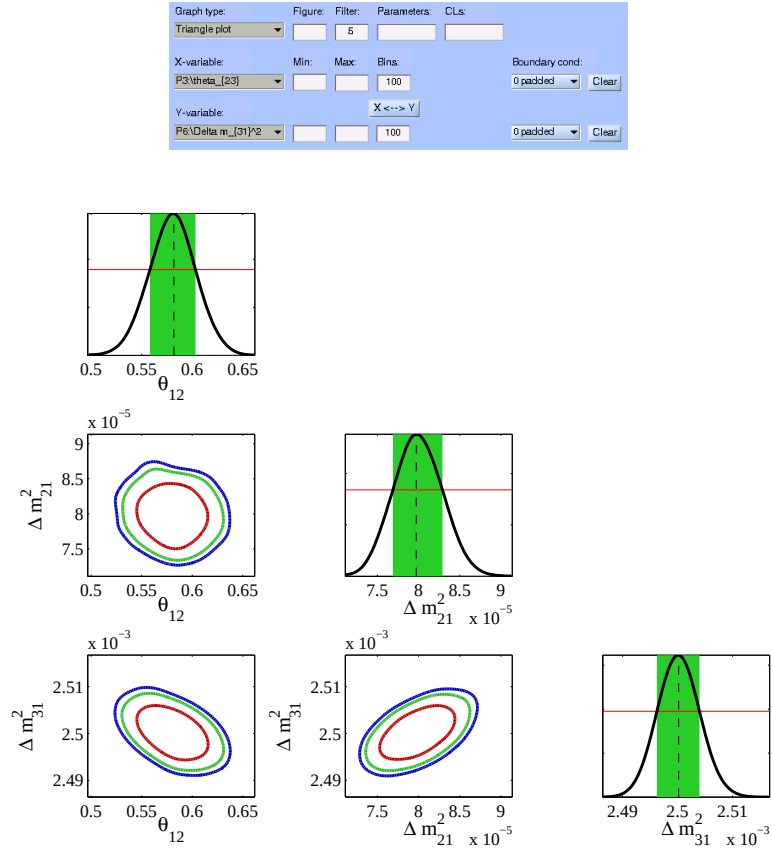


Figure 3.8: A triangle plot for the ISS neutrino factory simulation, as well as the GUI settings that produced it.

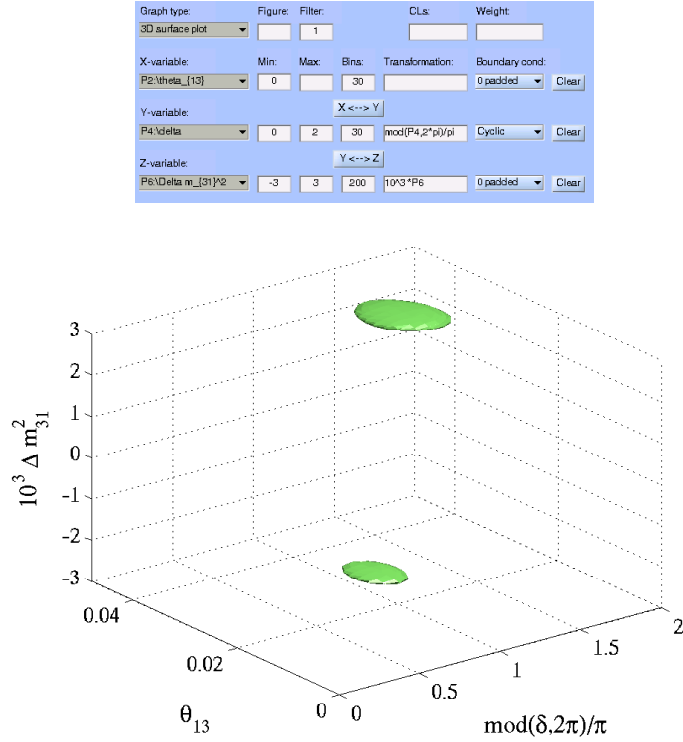


Figure 3.9: A 3D surface plot showing the degenerate solutions from the beta-beam simulation described in Sec. 2.4.2, as well as the GUI settings that produced it. See Sec. 3.3 for more details.

3.2.7 3D surface plots

The 3D surface plots are essentially the three-dimensional equivalents of the 2D counts contour plots and are created in exactly the same way. In Fig. 3.9, we present a 3D surface plot showing the two degeneracies of the example in Sec. 2.4.2. This plot type has the same parameter fields as its two-dimensional counterpart. However, note that three-dimensional filtering and binning requires quite a bit of computations and may therefore take some time. Also note that it will usually only be possible to see the outer of the surfaces.

Since this plotting type is mainly intended to allow the user to visualize the correlation between different parameters, it does not include the possibility of exporting any data.

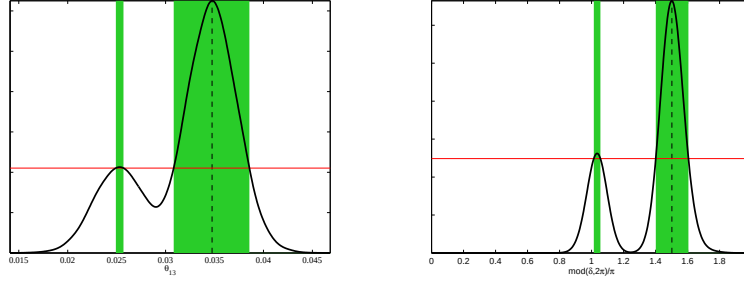


Figure 3.10: The 68 % confidence region plots of the degeneracy in θ_{13} and δ .

3.3 Results from the degenerate solution simulation

In Sec. 2.4.2, we showed how to treat cases with degenerate solutions. In particular, as an example, we have performed a simulation using a beta-beam with a known degeneracy where a flip of the sign of the atmospheric mass squared difference gives a solution which produces nearly as good a fit as the input values. The reason for choosing a scenario where the solutions are not completely degenerate is that we want to show that the Monte Carlo is able to produce the correct weights for the different degenerate solutions as well as exploring both of them properly. The actual determination of $|\Delta m_{31}^2|$ by the experiment is not very good, and thus we focus on the θ_{13} - δ parameter space, which is where the degeneracy is apparent. We already showed the degenerate solution in the example of the 3D surface plot and we now concentrate on the one- and two-dimensional plots, which are easier to interpret by inspection.

The one-dimensional confidence region plots are given in Fig. 3.10. From this figure, it is apparent that the input solution (essentially the best-fit) is more in line with the simulation results than the degenerate solution, which is only slightly allowed at the 68 % level in both θ_{13} and δ . Note that, although the degeneracies seem relatively close in the θ_{13} and δ parameters, there is actually a huge separation between them due to the flip of the sign of the atmospheric mass squared difference. The contour plot of the same degeneracy is shown in Fig. 3.11. While the degenerate solution is still smaller than the input solution, the difference now seems a little less pronounced. However, this is simply due to projection reasons, since the input solution both has a higher count rate and is broader, it will become more pronounced when summing the samples in one direction.

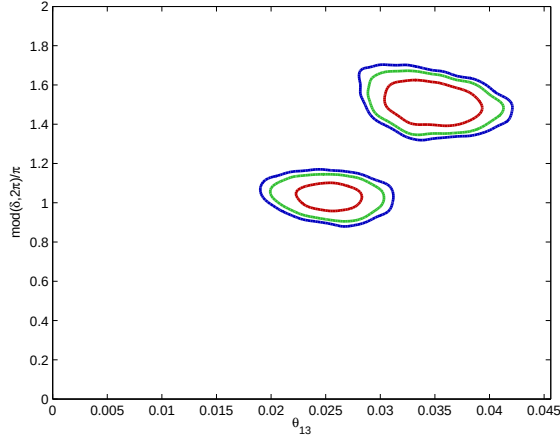


Figure 3.11: The 68 %, 95 %, and 99 % contours for the degeneracy in θ_{13} and δ .

3.4 Examples of bad convergence

As mentioned earlier, the convergence of the chains can be checked visually by studying some of the plots, in particular, the chain progression and scatter plots. For reference, we here present the results of the ISS neutrino factory simulation using a step size which is 50 times smaller than in our previous example. In this case, the convergence parameters are $R_{\theta_{23}} = 13.2623$ and $R_{\Delta m_{31}^2} = 1.0937$. The parameters used to produce the figures are the same as earlier in this chapter. The simulation contains four chains with 14000 samples each. In Fig. 3.12, we can clearly see the random walk structure of the chains as well as the jumps when changing chains, while Fig. 3.13 shows the sporadic coverage of the scatter plot, as well as the localization to only a few regions due to the small number of samples. Finally, in Fig. 3.14, we show the contours produced from this simulation. From this plot alone, it is quite apparent that the confidence regions are not what one would expect from a well-behaved simulation.

3.5 Notes on filtering

As mentioned earlier, both the confidence region plots and contour plots use Gaussian filters in order to smoothen the features of the simulation results. This is actually quite necessary in order to produce figures which do not look gobbled up, while maintaining a large number of bins. For example, Fig. 3.15 shows how Fig. 3.11 would appear without the filter. However, it is apparent that using a filter which is too wide will destroy features in the plot. Of course, what we really want to do is to destroy the features introduced by the random

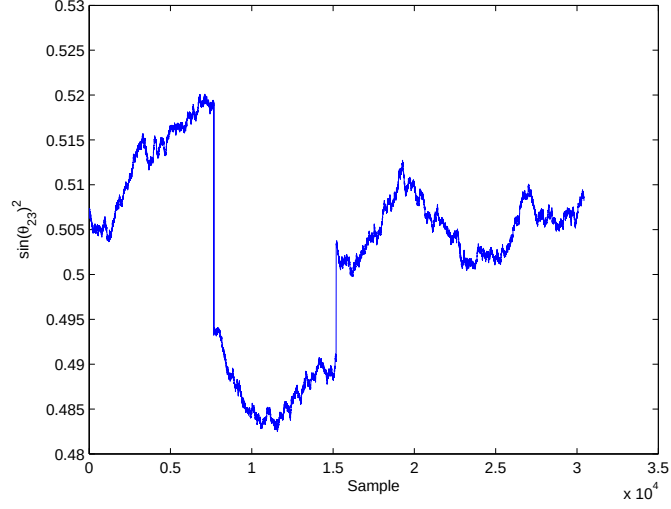


Figure 3.12: A 1D chain progression plot for $\sin^2(\theta_{23})$ for an ISS neutrino factory simulation using a step which is 50 times smaller than those used earlier.

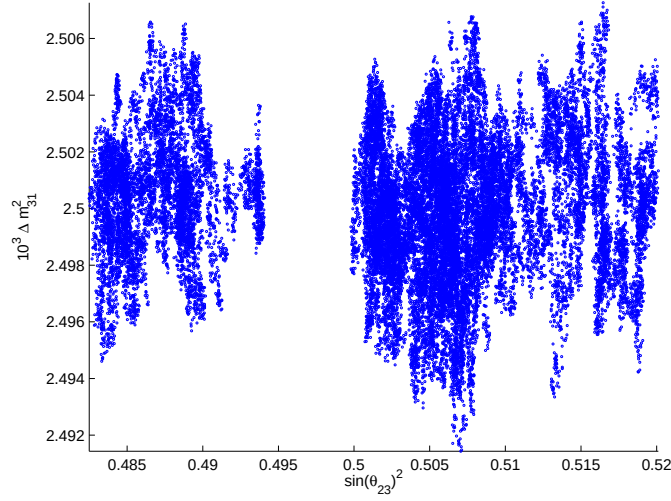


Figure 3.13: A scatter plot for $\sin^2(\theta_{23})$ and Δm^2_{31} for an ISS neutrino factory simulation using a step which is 50 times smaller than those used earlier.

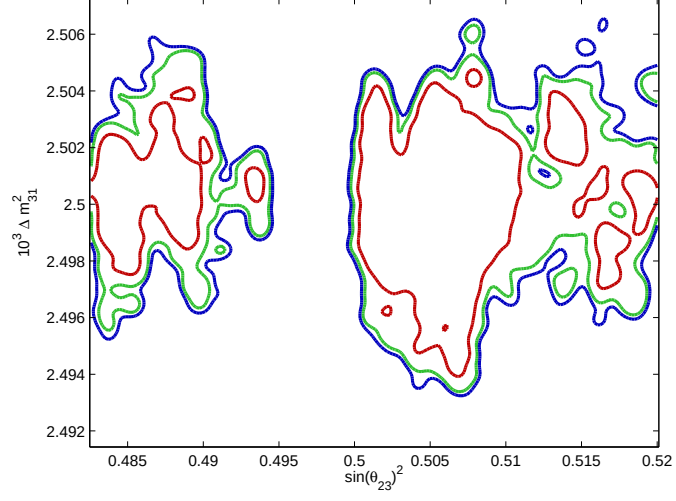


Figure 3.14: A contour plot for $\sin^2(\theta_{23})$ and Δm^2_{31} for an ISS neutrino factory simulation using a step which is 50 times smaller than those used earlier.

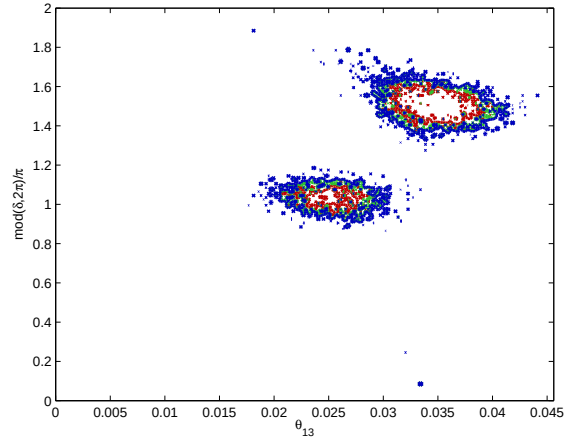


Figure 3.15: How Fig. 3.11 would appear without the Gaussian filter.

fluctuations while maintaining the features that are inherent of the physical setup. Thus, a valid question is: What number of bins and what filtering should one use?

Clearly, in order to properly reproduce a feature, we should use a bin size as well as a filter size which is relatively small compared to the typical size of the feature. Any feature smaller than the bin size will be lost upon binning, and any feature smaller than the filter size will be completely smoothed out into a Gaussian with the filter size.

The easiest way to produce a reliable figure is to simply pick an appropriate bin size and then to try different filter sizes. As the features of the graph are smoothened out by the filter, the results will always be conservative. Thus, the best filter size is one which makes the contours smooth and where decreasing the filter size does not significantly shrink the contours. If no such filter size can be found, this means that the simulation does not contain enough samples to reproduce the wanted features. This leaves the user with essentially three alternatives; using a relatively jagged graph, using a graph which is smooth but overestimates the sizes of the features in it, or rerunning the simulation with a larger number of samples.

Appendix A

Markov Chain Monte Carlo theory

Markov Chain Monte Carlos are a powerful tool which can be used to sample multi-dimensional probability distributions. Essentially, this is achieved by creating a Markov Chain that has the desired distribution as an equilibrium distribution. In particular, the method used in the **MonteCUBES** C library is the Metropolis–Hastings algorithm. In this chapter, we will describe this algorithm, as well as how it is implemented into **MonteCUBES**.

A.1 The Metropolis–Hastings sampling algorithm

Suppose that we have a probability density distribution $\mathcal{P}(x)$, which depends on the set of parameters x , and that we wish to construct a set of samples of this distribution through Monte Carlo methods. The Metropolis–Hastings (MH) algorithm performs this sampling through a random walk in the parameter space. In order to actually sample the distribution properly, this random walk must have the desired distribution as its equilibrium distribution. In the MH algorithm, this is achieved through the following steps:

1. To generate x_{n+1} , create a new random point in parameter space $\hat{x}$ according to the known jump probability function $\mathcal{W}(x_n \rightarrow \hat{x})$.
2. Compute the probability densities $\mathcal{P}(x_n)$ and $\mathcal{P}(\hat{x})$.
3. Pick a random number p from a uniform distribution in the interval $[0, 1]$.
If

$$p < \min \left(1, \frac{\mathcal{P}(\hat{x})\mathcal{W}(\hat{x} \rightarrow x_n)}{\mathcal{P}(x_n)\mathcal{W}(x_n \rightarrow \hat{x})} \right) = P(x_n \rightarrow \hat{x}),$$

then put $x_{n+1} = \hat{x}$. Otherwise, put $x_{n+1} = x_n$.

4. Repeat this procedure until the appropriate number of samples have been reached.

In order to see why this procedure has $\mathcal{P}(x)$ as its equilibrium distribution, let us consider the equilibrium condition of detailed balance

$$\mathcal{R}(x \rightarrow y) = \mathcal{R}(y \rightarrow x), \quad (\text{A.1})$$

where $\mathcal{R}(x \rightarrow y)$ is the transition rate from x to y . Since $\mathcal{R}(x \rightarrow y)$ is simply given by the product of $\mathcal{P}(x)$ (describing the probability to be in x) and $\mathcal{W}(x \rightarrow y)$ (the rate of transition from x to y given that we are starting in x), the detailed balance condition simply boils down to

$$\mathcal{P}(x)\mathcal{W}(x \rightarrow y) = \mathcal{P}(y)\mathcal{W}(y \rightarrow x). \quad (\text{A.2})$$

Thus, in order for our algorithm to sample $\mathcal{P}(x)$ as quickly as possible, we must make sure that it fulfills detailed balance while maximizing the probability of actually making a jump. Since the probability of accepting a test step is $P(x_n \rightarrow \hat{x})$ it follows that, at equilibrium,

$$\begin{aligned} \mathcal{R}(x \rightarrow y) &= \mathcal{P}(x)\mathcal{W}(x \rightarrow y)P(x \rightarrow y) \\ &= \mathcal{P}(x)\mathcal{W}(x \rightarrow y) \min \left(1, \frac{\mathcal{P}(y)\mathcal{W}(y \rightarrow x)}{\mathcal{P}(x)\mathcal{W}(x \rightarrow y)} \right) \\ &= \min (\mathcal{P}(x)\mathcal{W}(x \rightarrow y), \mathcal{P}(y)\mathcal{W}(y \rightarrow x)) \\ &= \mathcal{R}(y \rightarrow x), \end{aligned} \quad (\text{A.3})$$

where the last step follows from the symmetry of the expression. Since we have maximized the acceptance rate in one direction ($P(x \rightarrow y)$ cannot be larger than one), it must also be maximized in the other direction.

Essentially, we are free to choose any transition function $\mathcal{W}(x \rightarrow y)$. However, a very important special case, which is the one currently implemented in **MonteCUBES**, is the choice of the original Metropolis sampling algorithm, namely $\mathcal{W}(x \rightarrow y) = \mathcal{W}(y \rightarrow x)$. With this condition, the acceptance probability is simply the minimum of one and the ratio between $\mathcal{P}(\hat{x})$ and $\mathcal{P}(x_n)$. A future version of **MonteCUBES** may include the possibility of setting a user-defined transition function.

A.1.1 Convergence and finer details

Clearly, as the Monte Carlo chain test steps and acceptance criteria depend on the current step of the chain, there will in general be a correlation between subsequent steps within the same chain. In order for this correlation to be erased, it is necessary to create enough samples within each chain and put up criteria of convergence. The convergence criteria implemented in **MonteCUBES** are based on comparing the variances within each chain with the variance of the entire sample. In addition, in order to get rid of any dependence on the initial conditions, the first parts of the chains should be ignored. The process of ignoring the first part of the chains is known as *burning* and **MonteCUBES** has two different ways of implementing this:

1. Static burning. This method simply burns a predefined number of samples in each chain. It can be used when the user has a good grasp of how many steps must be taken between two samples before their correlation goes to zero. Since the sampling should not depend on the starting point, the final set of samples should not be correlated with the initial condition.
2. Dynamic burning. This method burns the initial half of the produced samples and then tests for convergence in the second half. If the chains have not converged, then the number of total samples is doubled and the same procedure is repeated. Effectively, when a set of samples is rejected, it and the samples burned before it become the new burn for the next set of samples. The positive aspect of using dynamic burning is that the user does not need to know anything about for how many steps the samples are correlated. The negative aspect is that half of the chains will be burned, although the number of steps before correlation is lost may be smaller than the number of steps necessary to reach convergence.

In addition to implementing these run-time burning conditions, **MonteCUBES** always stores the full chains, saving the burn lengths into the summary files. The effective burn length in a simulation can then be changed when using the **MonteCUBES Matlab GUI** to plot the results.

The convergence checks in a **MonteCUBES** simulation proceeds in the following manner [6]. Suppose that we have generated M chains, each containing N samples after the burned samples have been removed (in the case of dynamic burn, this would mean that the chains contain $2N$ samples in total). We then separately test for convergence for each parameter x that is allowed to vary in the simulation. By x_i^j , we denote the i th sample of x in chain j . Furthermore, we compute the mean of chain j

$$\bar{x}^j = \frac{1}{N} \sum_{i=1}^N x_i^j \quad (\text{A.4})$$

and the total mean

$$\bar{x} = \frac{1}{NM} \sum_{j=1}^M \sum_{i=1}^N x_i^j = \frac{1}{M} \sum_{j=1}^M \bar{x}^j. \quad (\text{A.5})$$

We now compute the variance in each chain as

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_i^j - \bar{x}^j)^2. \quad (\text{A.6})$$

A lower bound W on the variance in the complete set of samples is then given by the mean of the chain variances, *i.e.*,

$$W = \frac{1}{M} \sum_{j=1}^M \sigma_j^2 = \frac{1}{M(N-1)} \sum_{j=1}^M \sum_{i=1}^N (x_i^j - \bar{x}^j)^2. \quad (\text{A.7})$$

An upper bound on the same quantity can be constructed if we compute the variance between chains B , namely

$$B = \frac{1}{M-1} \sum_{j=1}^M (\bar{x}^j - \bar{x})^2, \quad (\text{A.8})$$

and the upper bound is

$$W' = \frac{N-1}{N} W + \frac{B}{M} (M+1). \quad (\text{A.9})$$

The parameter R used for the convergence check is then the ratio of these two estimates

$$R = \frac{N-1}{N} + \frac{B}{WM} (M+1). \quad (\text{A.10})$$

In practice, R is simply a measure of how well the variance in each chain corresponds to the total variance. If the convergence of all chains is good, then we expect this parameter to be close to one. However, if one or more chains have only sampled part of the target distribution, then the variance within the chains will be smaller than the variance in the complete sample and R will be significantly larger. In particular, this will be the case when there are degeneracies and different chains fall into different degeneracies. Each chain will then sample one of the degenerate solutions and the variance within the chains will be small compared to the variance in the complete set of samples. Since the distance between the degeneracies will in general be large, $\bar{x}^j$ will not be close to $\bar{x}$.

If the **MonteCUBES** C library is processing a chain with extremely large convergence parameters after a large number of steps, then it will warn the user of this and suggest using the built-in degeneracy solver in order to resolve this. *Warning!* While different chains may fall into different degeneracies and the **MonteCUBES** degeneracy solver is designed to allow chains to jump between different degeneracies, it is by no means implied that no degeneracies exist if they do not. Well separated degeneracies will not be reached if the simulations are started close to only one of the degeneracies.

A.2 Usage and interpretation of the methods from **GLoBES**

Since **MonteCUBES** is using **GLoBES** methods in order to find the likelihood ratios, we should devote some time to introducing how this is done. In order to discuss this properly, we first need to mention Bayes' theorem and its interpretation for discriminating among different models and/or parameter values. The statement of Bayes' theorem is

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \quad (\text{A.11})$$

where $P(A|B)$ is the conditional probability of A given B . In our case, we let A be the set of model parameters θ and B be the (actual or simulated) data points $\mathbf{D}$. The probability $P(\theta|\mathbf{D}) = \mathcal{P}_{\mathbf{D}}(\theta)$ is essentially the probability density that we wish to sample and gives the relative likelihoods of the parameter values given the data $\mathbf{D}$. The probability $P(\mathbf{D}|\theta) = \mathcal{L}_{\mathbf{D}}(\theta)$ is the likelihood of actually making the measurement $\mathbf{D}$ if the true parameters are θ , while $P(\theta) = \pi(\theta)$ is a prior function, which inputs our pre-knowledge of the theory parameters θ . Finally, $P(\mathbf{D}) = \mathcal{C}$ is interpreted as the probability of actually getting the measurement $\mathbf{D}$ within the given theory, *i.e.*,

$$\mathcal{C} = \int \mathcal{L}_{\mathbf{D}}(\theta)\pi(\theta)d\theta, \quad (\text{A.12})$$

which apparently does not depend on the model parameters, since they are integrated out. Since the Metropolis–Hastings algorithm is only using the ratios of the probability density functions, it is therefore equivalent to insert $\mathcal{P}_{\mathbf{D}}(\theta)$ and $\mathcal{L}_{\mathbf{D}}(\theta)\pi(\theta)$ as the probability density.¹

A.2.1 The GLoBES χ^2 as the log-likelihood

In order to compute how well a given data set $\mathbf{D}$ fits a specific set of model parameters θ , GLoBES uses the logarithm of the likelihood ratio

$$\lambda(\theta) = \prod_i f(n_i(\theta), \bar{n}_i) / f(\bar{n}_i, \bar{n}_i), \quad (\text{A.13})$$

where $n_i(\theta)$ is the predicted number of events in bin i given the parameters θ , $\bar{n}_i$ is the actual number of events in bin i , and $f(n, m)$ is the likelihood of m events given a Poisson distribution with mean n , it is given by

$$-2 \ln[f(n, m)/f(m, m)] = 2[n - m + m \ln \left(\frac{m}{n} \right)]. \quad (\text{A.14})$$

Again, the denominator is a normalizing constant and does not affect the ratios of the probability densities in the Monte Carlo. The nominal GLoBES χ^2 function is defined as²

$$\bar{\chi}^2(\theta) = -2 \ln \lambda(\theta) = 2 \sum_i \left[n_i(\theta) - \bar{n}_i + \bar{n}_i \ln \left(\frac{\bar{n}_i}{n_i(\theta)} \right) \right] \quad (\text{A.15})$$

and systematic errors are then taken into account using the pull method with Gaussian systematics to obtain the final χ^2 (see the GLoBES manual for details), which we will denote by $\chi^2(\theta)$. The actual probability density sampled by MonteCUBES is

$$\mathcal{P}(\theta) = \lambda(\theta)\pi(\theta) = \exp \left[-\frac{\chi^2(\theta)}{2} \right] \exp \left[-\frac{\chi_P^2(\theta)}{2} \right], \quad (\text{A.16})$$

¹The constant $\mathcal{C}$ can essentially be used to discriminate among different models.

²Note that this does not follow a χ^2 distribution except in the limit of large samples.

where $\chi_P^2(\boldsymbol{\theta}) = -2 \ln \pi(\boldsymbol{\theta})$ (with a slight abuse of notation, this quantity is referred to as the prior, although it is actually related to the logarithm of the prior function). The central quantity used in **MonteCUBES** is $\hat{\chi}^2 = \chi^2 + \chi_P^2$, this is the actual number that will be stored along with the samples in the **MonteCUBES** output files (see Sec. 2.3).

A.2.2 Notes on priors

Obviously, a given parametrization of a model is not necessarily the only one and the parametrization used to implement the physics of the model may not be the same as the one which is most illuminating. For example, **GLOBES** implements the mixing angles and the CP -phase as the parameters of the neutrino oscillation theory, while it is common to plot the results as a function of $\sin^2$ of the angles (or multiples of the angles). Since this transformation may not preserve volumes in the parameter space, it will also affect the probability density. Let us assume that we make the transformation $\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}'$. For the relation $\mathcal{P}(\boldsymbol{\theta})d\boldsymbol{\theta} = \mathcal{P}'(\boldsymbol{\theta}')d\boldsymbol{\theta}'$ to hold, we must have

$$\mathcal{P}'(\boldsymbol{\theta}') = \mathcal{P}(\boldsymbol{\theta})|\mathcal{J}(\boldsymbol{\theta})|, \quad (\text{A.17})$$

where $\mathcal{J}(\boldsymbol{\theta})$ is the Jacobian of the transformation. In order to preserve the expression for the probability density in terms of the likelihood and prior, we must have

$$\pi'(\boldsymbol{\theta}') = \pi(\boldsymbol{\theta})|\mathcal{J}(\boldsymbol{\theta})|. \quad (\text{A.18})$$

Thus, the actual value of the prior function is parametrization dependent and the prior does not have the same expression for different parametrizations. In particular, this means that assuming a flat prior (which is essentially saying that any parameter values are equally likely) in one parametrization will still give a non-flat prior in some other parametrization. The **MonteCUBES** GUI includes a weight function so that the user can essentially change the prior by hand post-simulation. This can be useful, *e.g.*, when making plots where $\sin^2(2\theta)$ is used and the user does not want to rewrite the internal **GLOBES** or **MonteCUBES** prior functions. The implied prior in the $\sin^2(2\theta)$ space will give a pile-up of samples near $\sin^2(2\theta) = 1$ but multiplication of the number of samples by $|d\sin^2(2\theta)/d\theta| = |2\sin(4\theta)|$ will give a flat prior in this space (assuming the original prior was flat in θ).³ The multiplication of the sample number in the GUI is done on a sample-by-sample basis.

A.2.3 Interpreting the results

All of the **MonteCUBES** GUI methods that provide a confidence region of some sort (*i.e.*, 1D histogram, 1D confidence region, 2D contours, and 3D surface plots) build upon the same principle. Initially, the region for which to make the plot is divided into bins and the (weighted) number of samples in each bin is computed after which any filter introduced to suppress statistical fluctuations is applied.

³Note that the region close to $\sin^2(2\theta)$ will still be sampled in greater detail.

The plotted regions includes all bins with a total number of samples larger than N , where N is chosen such that the region includes a given percentage α of the total number of samples. The resulting region is the smallest region containing the fraction α of the samples. In the case of the 1D confidence region, there is also a line denoting the point of maximum probability density, which is simply the bin with the largest number of samples.

A.3 The MonteCUBES degeneracy solver

A common feature of many setups for neutrino oscillation experiments is the appearance of solutions which are degenerate. In fact, degenerate solutions are in many cases the source of the main uncertainties predicted for different oscillation parameters in several experiments. In this section, we will describe how the MonteCUBES degeneracy solver works from a theoretical standpoint. For an example on how to use it, see Sec. 2.4.2.

If the distribution that we are sampling have well separated degeneracies, then we are faced with a problem when choosing the appropriate step sizes for our Monte Carlo simulations. Assuming that we only take Gaussian steps of a given average size, there are two possible choices:

1. *Degeneracy sized steps.* If we use step sizes comparable to the size of each degenerate solution, then the solution that we happen to fall into will be sampled very well. However, the probability of jumping between the degeneracies will be severely suppressed. This is due to the fact that, in order to jump, the chains must at some point pass through the region which should not contain so many samples. If $\Delta\chi^2$ is the typical difference in χ^2 between the lowest point of the wall and the minimum of the degeneracy, then the simulation should sample the minimum point about $\exp(\Delta\chi^2/2)$ times for each time that the lowest wall point is sampled. If $\Delta\chi^2 = \mathcal{O}(200)$, which is not at all unreasonable, then this ratio is $\mathcal{O}(10^{43})$. Adding the fact that, in order to be sure that the chains have sampled the degenerate solutions with proper weights, the chains should jump a large number of times. A simulation that properly samples the degeneracies with this choice of step length would have to run for a very long time.
2. *Degeneracy difference sized steps.* If we instead use steps of a size which is comparable to the size of the step needed to jump between the degeneracies without having to sample points in between, then the chains will hardly ever jump. This would result in the degeneracies being sampled with about the correct weights. However, it would have the drawback of not exploring each individual degeneracy in any detail. Thus, this choice also seems like a bad idea.

A.3.1 Chain heating

One way of dealing with degeneracies is to *heat* the sampled distribution. Instead of sampling the actual distribution, we sample the distribution $\mathcal{P}_T(x) = \mathcal{P}(x)^{1/T}$, where $T > 1$ is the temperature parameter. The distribution $\mathcal{P}_T(x)$ can, depending on T , be significantly flatter than the original distribution and the relative weights of the samples will clearly change. Thus, in order to interpret the results, we somehow need to relate the sample from $\mathcal{P}_T(x)$ with a sample of $\mathcal{P}(x)$. This posterior process of relating a heated chain with the original distribution is known as *cooling*.

Once the heated chains have been produced, we therefore need to find a method of using posterior cooling. Since the distributions we are sampling are continuous, the number of times a point is sampled is not necessarily proportional to $\mathcal{P}(x)$ or $\mathcal{P}_T(x)$ (imagine that we already have a way of picking random samples from $\mathcal{P}(x)$, then we could use $\mathcal{W}(x \rightarrow y) = \mathcal{P}(y)$ and have a transition probability of one, meaning that each point would essentially only be sampled once). Instead, we are interested in the sampling density and it is therefore a valid point to ask how much more or less likely it is that the point would have been sampled given a cooler chain. Since we are anyway computing $\mathcal{P}(x)$ at the sampling points, the way of doing this is to use the transformation

$$n_i \propto n_i^T \frac{\mathcal{P}(x_i)}{\mathcal{P}_T(x_i)}, \quad (\text{A.19})$$

where n_i is number of samples for the cooled distribution, n_i^T is the number of samples for the heated distribution, and x_i is the sample point. Since the actual proportionality constant is irrelevant for our purposes, we may as well put it to one. The reason to use this transformation is that point in the vicinity of x_i should be sampled with a rate proportional to $\mathcal{P}(x_i)$ when using the cooled distribution, while they are sampled with a rate proportional to $\mathcal{P}_T(x_i)$. Thus, multiplying with the fraction between these distributions generates the correct relative sampling frequencies.

The method of heating chains also has some problems. The largest of these being the fact that our simulations will spend a lot of time in regions where the actual probability density is quite small. If the region in between the degeneracies contains points with a very small probability density and the degeneracies are well separated, then the chains will have to be heated by a very large amount, meaning that the degenerate solutions themselves will not be that well explored.

A.3.2 Degenerate steps

Another way of dealing with degeneracies is based on the fact that the Metropolis–Hastings sampling algorithm allows for quite general transition functions. If the user knows where the degeneracies are located with respect to each other, then it is possible to construct a transition function which takes this into account. This way of solving degeneracies is also integrated into the `MonteCUBES` C library. This is implemented in such a way that the test steps in the Markov

Chains, in addition to the random Gaussian step, takes a step in the degeneracy direction with a predefined probability p . If $\mathcal{W}_0(x \rightarrow y)$ is the original transition function, the new transition function is given by

$$\mathcal{W}(x \rightarrow y) = (1 - p)\mathcal{W}_0(x \rightarrow y) + \frac{p}{2}[\mathcal{W}_0(x \rightarrow y - d) + \mathcal{W}_0(x \rightarrow y + d)], \quad (\text{A.20})$$

where d is the difference vector in parameter space between the degenerate solutions. The factor of $1/2$ in the second term comes from taking the degeneracy step in two different directions with equal probability. It is easy to check that $\mathcal{W}(x \rightarrow y)$ is symmetric if $\mathcal{W}_0(x \rightarrow y)$ is. This way of solving degeneracies can be likened with digging a tunnel between the degenerate solutions. The specific choice of transition function makes it possible for the Markov Chains to pass from one degenerate solution to the other without having to pass the ridges of small probability densities.

A.3.3 Which method do I use?

A good question when dealing with degeneracies is what method should be used to resolve them. Clearly, both solutions have both up- and downsides. While the chain heating method will mean sacrificing computing time for exploring very low-probability parts of the parameter space, the degeneracy step method demands that the user already knows where the degenerate solutions are located. Thus, the method employed depends greatly on what the user wants to know. If the user has no pre-knowledge on where the degenerate solutions are located, then it may be a good idea to start out by heating the chains in order to find the degeneracies. If the chains do not need to be heated by a large amount (*i.e.*, the degeneracies are relatively close), then this may be sufficient to also explore the structures of the degeneracies themselves. However, if the degeneracies are well separated, then it may be useful to run some heated chains first. If these runs discover degeneracies, but do not resolve them with proper accuracy, then the information on the distance between the degeneracies can be implemented in a run using the degenerate step degeneracy solver in order to make a more detailed exploration of the degenerate solutions.

A.3.4 The temperature lowering degeneracy finder

One way of dealing with the dilemma of which method to use to treat degeneracies is to use both. This is implemented in **MonteCUBES** through using a heating method in order to locate the degeneracies and then using the appropriate degeneracy steps between these degenerate solutions in order to explore the regions around them properly.

This degeneracy finder works by running several chains starting at a high temperature $T_h \gg 1$ (so that the walls between the degeneracies can be passed). When a sufficient number of Monte Carlo steps have been taken, so that the chains have reached equilibrium at the high temperature, the temperature is lowered and the chains are run for the same number of steps at this lower

temperature (note that the chains are not run to equilibrium at the lower temperatures, since we want to find the degeneracies). This process is repeated until the chains are at a very low temperature $T_l \ll 1$, where the degeneracies are extremely pronounced. The final samples in the chains will then be located close to the different local minima. Thus, the degeneracy finder checks the resulting chain endpoints for different minima (two endpoints are considered as belonging to the same minima if they are less than one Monte Carlo step size at $T = 1$ away from each other).

The step sizes used by the degeneracy finder vary from temperature to temperature. Essentially, the degeneracy finder uses the same steps as the main Markov Chain Monte Carlo at $T = 1$. When $T \neq 1$, the steps sizes are multiplied by $\sqrt{T}$ in order to accommodate for the flatter distribution.

Note that it is not guaranteed that each degenerate solution will be explored by this method. There are essentially two things that can go wrong:

1. The original temperature is not high enough. If this is the case, then the degeneracy finder will still not be able to pass between the degeneracies and will only find degeneracies between which the temperature is sufficient.
2. Pure chance. There is always the pure statistical chance that one degenerate solution will be missed. The probability of this depends on the number of chains. Clearly, if the number of chains is smaller than the number of degeneracies, the probability is one. Therefore, we recommend that you run a number of chains which is about an order of magnitude larger than the expected number of degeneracies. In Fig. A.1, we show the probability of missing one or more of the degenerate solutions as a function of the number of chains and the number of degeneracies.

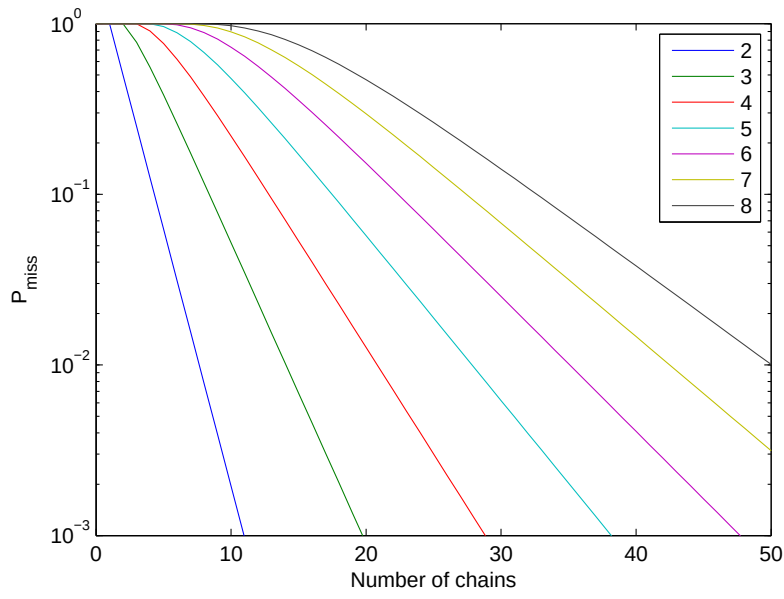


Figure A.1: The probability of missing one or more of the degenerate solutions under the assumption that the chains fall into each degeneracy with equal probability. The labels denote the total number of degeneracies.

Appendix B

The NonUnitarity Engine (NUE)

The NonUnitarity Engine (NUE) is a probability engine implementing the physics of a non-unitary lepton mixing matrix into GLoBES. The NUE is installed along with the MonteCUBES C library. However, the methods of NUE are included in the header file `nue_probability.h` rather than in `montecubes.h`, although these will be located in the same directory after installation. You can also use the NUE independently from the rest of MonteCUBES. This appendix describes the functionality that is added when including the header file through the command:

```
#include <montecubes/nue_probability.h>
```

B.1 Theory of a non-unitary mixing matrix

A non-unitary lepton mixing matrix in the CC interaction between neutrinos and charged leptons is the generic feature of models involving extra degrees of freedom that can mix with either of the lepton components. In particular, in the popular type-I seesaw models accommodating the smallness of neutrino masses through the addition of heavy fermionic singlets (right-handed neutrinos), these extra degrees of freedom will mix with the light active neutrinos giving rise to a larger mixing matrix than the standard three by three one. The three by three sub-matrix involving the mixing between the light mass eigenstates accessible at low energies and the three active flavour eigenstates, will in general not be unitary. In standard seesaw models these unitarity violation is expected to be unobservably small. On the hand, these violations are induced by an independent, lepton number conserving operator than the one that generates neutrino masses. The smallness of the neutrino mass can then be naturally accommodated through a slightly broken lepton number symmetry, as in the

inverse or double seesaw models, with large testable deviations from unitarity of the lepton mixing matrix.

A convenient way of parameterizing the effects of non-unitary mixing in neutrino oscillations is splitting the general non-unitary matrix N as the product of an hermitian times a unitary matrix $N = (1 + \varepsilon)U$ where $\varepsilon^\dagger = \varepsilon$. Since strong constraints can be derived on the unitarity deviations through electroweak decays, ε should be a small perturbation and $U \simeq U_{PMNS}$. We will adopt this parametrization adding the extra six modulus and three phases included in the hermitian ε to the standard parameters of the unitary part of the general mixing matrix.

B.2 API definitions

This section contains descriptions of the methods declared in the NUE header file, as well as a list of and explanations for the different constants that are defined in it.

B.2.1 Methods

`void nue_fixAllNU (glb_projection p)`

This method sets all projection flags for the non-unitarity parameters in a GLoBES projection to `GLB_FIXED`.

`p` – The GLoBES projection to change.

Returns:

`void`

`void nue_freeAllNU (glb_projection p)`

This method sets all projection flags for the non-unitarity parameters in a GLoBES projection to `GLB_FREE`.

`p` – The GLoBES projection to change.

Returns:

`void`

`void nue_fixPhases (glb_projection p)`

This method sets all projection flags for the non-unitarity phase parameters in a GLoBES projection to `GLB_FIXED`.

`p` – The GLoBES projection to change.

Returns:

void

```
void nue_freePhases (glb_projection p)
```

This method sets all projection flags for the non-unitarity phase parameters in a GLoBES projection to GLB_FREE.

p – The GLoBES projection to change.

Returns:

void

```
void nue_fixEpss (glb_projection p)
```

This method sets all projection flags for the non-unitarity absolute value parameters in a GLoBES projection to GLB_FIXED.

p – The GLoBES projection to change.

Returns:

void

```
void nue_freeEpss (glb_projection p)
```

This method sets all projection flags for the non-unitarity absolute value parameters in a GLoBES projection to GLB_FREE.

p – The GLoBES projection to change.

Returns:

void

```
void nue_fixEps (glb_projection p, int n, int m)
```

This method sets the projection flag for the absolute value of the non-unitarity parameter in the `n`th row and `m`th column to GLB_FIXED. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FIXED, NUE_EPS_nm)
```

p – The GLoBES projection to change.

n – The row of the flag to set. Should be 1, 2 or 3.

m – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nue_freeEps (glb_projection p, int n, int m)
```

This method sets the projection flag for the absolute value of the non-unitarity parameter in the `n`th row and `m`th column to `GLB_FREE`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FREE, NUE_EPS_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nue_fixPhi (glb_projection p, int n, int m)
```

This method sets the projection flag for the phase of the non-unitarity parameter in the `n`th row and `m`th column to `GLB_FIXED`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FIXED, NUE_PHI_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nue_freeEps (glb_projection p, int n, int m)
```

This method sets the projection flag for the phase of the non-unitarity parameter in the `n`th row and `m`th column to `GLB_FREE`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FREE, NUE_PHI_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

`void`

`void nue_toPhysical (glb_params toTransf, void* user_data)`

This is a physical transformation for usage with `MonteCUBES`. It simply puts the phases between $-\pi$ and π and makes sure that the absolute values are positive. In addition, it calls the standard `MonteCUBES` function to make sure that the standard parameters are in the physical region.

`toTransf` – The `GLOBES` parameter vector to transform. The entries of this vector will be changed by the function.

`user_data` – Ignored. Included for compliance with the `MonteCUBES` standard.

Returns:

`void`

`double nue_prior (glb_params p, void* user_data)`

Currently equivalent to the standard `GLOBES` prior.

`p` – The parameter vector for which to compute the prior.

`user_data` – Ignored. Included for compliance with the `GLOBES` standard.

Returns:

The value of the prior.

`void nue_setAllNU (glb_params p, double v)`

This method can be used to set all of the non-unitarity parameters in a `GLOBES` parameter vector to a certain value.

`p` – The `GLOBES` parameter vector in which to set the values.

`v` – The value to use for the parameters being set.

Returns:

`void`

```
void nue_setEpss (glb_params p, double v)
```

This method can be used to set all of the absolute values of the non-unitarity parameters in a GLoBES parameter vector to a certain value.

p – The GLoBES parameter vector in which to set the values.

v – The value to use for the parameters being set.

Returns:

void

```
void nue_setPhis (glb_params p, double v)
```

This method can be used to set all of the phases of the non-unitarity parameters in a GLoBES parameter vector to a certain value.

p – The GLoBES parameter vector in which to set the values.

v – The value to use for the parameters being set.

Returns:

void

```
int nue_init_probability_engine ()
```

The method that should be called to initiate the NUE.

Returns:

0

```
int nue_free_probability_engine ()
```

The method that should be called when freeing the memory allocated by the NUE.

Returns:

0

```
int nue_set_oscillation_parameters (glb_params p, void* user_data)
```

Sets the oscillation parameters that are used by the NUE to those specified in a GLoBES parameter vector.

p – The GLoBES parameter vector to use.

`user_data` – Ignored. Included for compliance with the GLoBES standard.

Returns:

0

```
int nue_get_oscillation_parameters (glb_params p, void* user_data)
```

Stores the oscillation parameters that are used by the NUE in the specified GLoBES parameter vector.

`p` – The GLoBES parameter vector to use.

`user_data` – Ignored. Included for compliance with the GLoBES standard.

Returns:

0

```
int nue_probability_matrix (double P[3][3], int cp_sign, double
                             E, int psteps, const double *length, const double *density, double
                             filter_sigma, void *user_data)
```

The method to compute the matrix of oscillation probabilities using the NUE.

`P` – A two-dimensional double array to store the oscillation probabilities.

`cp_sign` – Flag for neutrinos or anti-neutrinos, as per the GLoBES manual.

`E` – The neutrino energy, as per the GLoBES manual.

`psteps` – The number of steps, as per the GLoBES manual.

`length` – The length of the steps, as per the GLoBES manual.

`density` – The density in the steps, as per the GLoBES manual.

`filter_sigma` – Ignored. At present, the NUE is only able to compute the non-filtered probabilities.

`user_data` – This should be an integer array containing two entries. These entries are flags telling the NUE if to normalize the probabilities (1) or not (0). The first entry of the array concerns the source normalization and the second entry the detector normalization.

Returns:

0

B.2.2 Constants

NUE_EPS_11, NUE_EPS_EE

Both of these constants are the same. They refer to the internal index of the non-unitary parameter ε_{ee} .

NUE_EPS_22, NUE_EPS_MM

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $\varepsilon_{\mu\mu}$.

NUE_EPS_33, NUE_EPS_TT

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $\varepsilon_{\tau\tau}$.

NUE_EPS_21, NUE_EPS_ME

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $|\varepsilon_{\mu e}|$.

NUE_EPS_31, NUE_EPS_TE

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $|\varepsilon_{\tau e}|$.

NUE_EPS_32, NUE_EPS_TM

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $|\varepsilon_{\tau\mu}|$.

NUE_PHI_21, NUE_PHI_ME

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $\arg(\varepsilon_{\mu e})$.

NUE_PHI_31, NUE_PHI_TE

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $\arg(\varepsilon_{\tau e})$.

NUE_PHI_32, NUE_PHI_TM

Both of these constants are the same. They refer to the internal index of the non-unitary parameter $\arg(\varepsilon_{\tau\mu})$.

NUE_TOT_NO

This constant has the value of the total number of oscillation parameters within the NUE.

NUE_EPSS

This constant has the value of the lowest index referring to a parameter describing the absolute values of the ε s.

NUE_PHIS

This constant has the value of the lowest index referring to a parameter describing the phase of an ε .

Appendix C

The non-Standard Interaction Event Generator Engine (nSIEGE)

The non-Standard Interaction Event Generator Engine (nSIEGE) is a probability engine implementing the physics of a non-standard matter interaction between background matter and neutrinos into GLoBES. The nSIEGE is installed along with the `MonteCUBES` C library. However, the methods of nSIEGE are included in the header file `nsi_probability.h` rather than in `montecubes.h`, although these will be located in the same directory after installation. You can also use the nSIEGE independently from the rest of `MonteCUBES`. This appendix describes the functionality that is added when including the header file through the command:

```
#include <montecubes/nsi_probability.h>
```

C.1 Theory of a non-standard matter interactions

Non-standard interactions (NSI) of neutrinos with matter are expected in a number of different extensions of the Standard Model (SM). Usually, the NSI are parametrized independently of the underlying theory through effective four-fermion operators

$$\mathcal{L}_{\text{NSI}} = -2\sqrt{2}G_F\varepsilon_{\alpha\beta}^{fP}[\bar{f}\gamma^\mu Pf][\bar{\nu}_\alpha\gamma_\mu P_L\nu_\beta], \quad (\text{C.1})$$

where f is a matter fermion and P is a left- or right-handed projector. In addition to these neutral-current-like (NC-like) processes, extensions of the SM

will in general also imply charged-current-like (CC-like) NSI, usually with some correlation between the couplings.¹ However, it has been common to study the flavor propagation effects induced by NC-like processes separately from the source and detector effects induced by the CC-like processes because of the lack of model independent relations between the two.

The effect that NSI have on the neutrino flavor propagation can be effectively described through making the substitution

$$H_{\text{matter}} = \sqrt{2}G_F N_e \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \longrightarrow \sqrt{2}G_F N_e \begin{pmatrix} 1 + \varepsilon_{ee} & \varepsilon_{\mu e}^* & \varepsilon_{\tau e}^* \\ \varepsilon_{\mu e} & \varepsilon_{\mu\mu} & \varepsilon_{\tau\mu}^* \\ \varepsilon_{\tau e} & \varepsilon_{\tau\mu} & \varepsilon_{\tau\tau} \end{pmatrix}, \quad (\text{C.2})$$

where N_e is the electron number density and the $\varepsilon_{\alpha\beta}$ are combinations of the $\varepsilon_{\alpha\beta}^{fP}$ s weighted properly for the relative abundance of the corresponding fermions. Note that the diagonal elements are real due to hermiticity. Just as in the case of unitarity violation (see App. B), this means that the new physics are parametrized through the introduction of an arbitrary hermitian matrix, which we again parametrize according to the modulus and phases, $\varepsilon_{\alpha\beta} = |\varepsilon_{\alpha\beta}| \exp(i\phi_{\alpha\beta})$. It should be noted that the parameters used here do *not* give the same matter interaction term as in the NUE, although we will use the same notation for the extra parameters.

C.2 API definitions

This section contains descriptions of the methods declared in the nSIEGE header file, as well as a list of and explanations for the different constants that are defined in it.

C.2.1 Methods

```
void nsi_fixAllNSI (glb_projection p)
```

This method sets all projection flags for the NSI parameters in a GLoBES projection to GLB_FIXED.

p – The GLoBES projection to change.

Returns:

void

```
void nsi_freeAllNSI (glb_projection p)
```

This method sets all projection flags for the NSI parameters in a GLoBES projection to GLB_FREE.

¹An example of this is the scheme of minimal unitarity violation, which is also implemented into MonteCUBES through NUE, see App. B.

p – The GLoBES projection to change.

Returns:

void

`void nsi_fixPhases (glb_projection p)`

This method sets all projection flags for the NSI phase parameters in a GLoBES projection to GLB_FIXED.

p – The GLoBES projection to change.

Returns:

void

`void nsi_freePhases (glb_projection p)`

This method sets all projection flags for the NSI phase parameters in a GLoBES projection to GLB_FREE.

p – The GLoBES projection to change.

Returns:

void

`void nsi_fixEpss (glb_projection p)`

This method sets all projection flags for the NSI absolute value parameters in a GLoBES projection to GLB_FIXED.

p – The GLoBES projection to change.

Returns:

void

`void nsi_freeEpss (glb_projection p)`

This method sets all projection flags for the NSI absolute value parameters in a GLoBES projection to GLB_FREE.

p – The GLoBES projection to change.

Returns:

void

```
void nsi_fixEps (glb_projection p, int n, int m)
```

This method sets the projection flag for the absolute value of the NSI parameter in the `nth` row and `mth` column to `GLB_FIXED`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FIXED, NSI_EPS_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nsi_freeEps (glb_projection p, int n, int m)
```

This method sets the projection flag for the absolute value of the NSI parameter in the `nth` row and `mth` column to `GLB_FREE`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FREE, NSI_EPS_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nsi_fixPhi (glb_projection p, int n, int m)
```

This method sets the projection flag for the phase of the NSI parameter in the `nth` row and `mth` column to `GLB_FIXED`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FIXED, NSI_PHI_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nsi_freeEps (glb_projection p, int n, int m)
```

This method sets the projection flag for the phase of the NSI parameter in the `n`th row and `m`th column to `GLB_FREE`. This is equivalent to:

```
glbSetProjectionFlag(p, GLB_FREE, NSI_PHI_nm)
```

`p` – The GLoBES projection to change.

`n` – The row of the flag to set. Should be 1, 2 or 3.

`m` – The column of the flag to set. Should be 1, 2 or 3.

Returns:

void

```
void nsi_toPhysical (glb_params toTransf, void* user_data)
```

This is a physical transformation for usage with `MonteCUBES`. It simply puts the phases between $-\pi$ and π and makes sure that the absolute values are positive. In addition, it calls the standard `MonteCUBES` function to make sure that the standard parameters are in the physical region.

`toTransf` – The GLoBES parameter vector to transform. The entries of this vector will be changed by the function.

`user_data` – Ignored. Included for compliance with the `MonteCUBES` standard.

Returns:

void

```
double nsi_prior (glb_params p, void* user_data)
```

Currently equivalent to the standard GLoBES prior.

`p` – The parameter vector for which to compute the prior.

`user_data` – Ignored. Included for compliance with the GLoBES standard.

Returns:

The value of the prior.

```
void nsi_setAllNSI (glb_params p, double v)
```

This method can be used to set all of the NSI parameters in a GLoBES parameter vector to a certain value.

p – The GLoBES parameter vector in which to set the values.

v – The value to use for the parameters being set.

Returns:

void

```
void nsi_setEpss (glb_params p, double v)
```

This method can be used to set all of the absolute values of the NSI parameters in a GLoBES parameter vector to a certain value.

p – The GLoBES parameter vector in which to set the values.

v – The value to use for the parameters being set.

Returns:

void

```
void nsi_setPhis (glb_params p, double v)
```

This method can be used to set all of the phases of the NSI parameters in a GLoBES parameter vector to a certain value.

p – The GLoBES parameter vector in which to set the values.

v – The value to use for the parameters being set.

Returns:

void

```
int nsi_init_probability_engine ()
```

The method that should be called to initiate the nSIEGE.

Returns:

0

```
int nsi_free_probability_engine ()
```

The method that should be called when freeing the memory allocated by the nSIEGE.

Returns:

0

```
int nsi_set_oscillation_parameters (glb_params p, void* user_data)
```

Sets the oscillation parameters that are used by the nSIEGE to those specified in a GLoBES parameter vector.

`p` – The GLoBES parameter vector to use.

`user_data` – Ignored. Included for compliance with the GLoBES standard.

Returns:

0

```
int nsi_get_oscillation_parameters (glb_params p, void* user_data)
```

Stores the oscillation parameters that are used by the nSIEGE in the specified GLoBES parameter vector.

`p` – The GLoBES parameter vector to use.

`user_data` – Ignored. Included for compliance with the GLoBES standard.

Returns:

0

```
int nsi_probability_matrix (double P[3][3], int cp_sign, double
    E, int psteps, const double *length, const double *density, double
    filter_sigma, void *user_data)
```

The method to compute the matrix of oscillation probabilities using the nSIEGE.

`P` – A two-dimensional double array to store the oscillation probabilities.

`cp_sign` – Flag for neutrinos or anti-neutrinos, as per the GLoBES manual.

`E` – The neutrino energy, as per the GLoBES manual.

`psteps` – The number of steps, as per the GLoBES manual.

`length` – The length of the steps, as per the GLoBES manual.

`density` – The density in the steps, as per the GLoBES manual.

`filter_sigma` – Ignored. At present, the nSIEGE is only able to compute the non-filtered probabilities.

`user_data` – Ignored. Included for compliance with the GLoBES standard.

Returns:

0

C.2.2 Constants

`NSI_EPS_11`, `NSI_EPS_EE`

Both of these constants are the same. They refer to the internal index of the NSI parameter ε_{ee} .

`NSI_EPS_22`, `NSI_EPS_MM`

Both of these constants are the same. They refer to the internal index of the NSI parameter $\varepsilon_{\mu\mu}$.

`NSI_EPS_33`, `NSI_EPS_TT`

Both of these constants are the same. They refer to the internal index of the NSI parameter $\varepsilon_{\tau\tau}$.

`NSI_EPS_21`, `NSI_EPS_ME`

Both of these constants are the same. They refer to the internal index of the NSI parameter $|\varepsilon_{\mu e}|$.

`NSI_EPS_31`, `NSI_EPS_TE`

Both of these constants are the same. They refer to the internal index of the NSI parameter $|\varepsilon_{\tau e}|$.

`NSI_EPS_32`, `NSI_EPS_TM`

Both of these constants are the same. They refer to the internal index of the NSI parameter $|\varepsilon_{\tau\mu}|$.

`NSI_PHI_21`, `NSI_PHI_ME`

Both of these constants are the same. They refer to the internal index of the NSI parameter $\arg(\varepsilon_{\mu e})$.

`NSI_PHI_31`, `NSI_PHI_TE`

Both of these constants are the same. They refer to the internal index of the NSI parameter $\arg(\varepsilon_{\tau e})$.

`NSI_PHI_32`, `NSI_PHI_TM`

Both of these constants are the same. They refer to the internal index of the NSI parameter $\arg(\varepsilon_{\tau\mu})$.

`NSI_TOT_NO`

This constant has the value of the total number of oscillation parameters within the nSIEGE.

`NSI_EPSS`

This constant has the value of the lowest index referring to a parameter describing the absolute values of the ε s.

NSI_PHIS

This constant has the value of the lowest index referring to a parameter describing the phase of an ε .

Appendix D

GNU General Public License

GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received.

You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under

applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the

absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.
- (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

- (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial,

industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional

permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- (d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- (e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- (f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License

(including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose

a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the

covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts

shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
```

```
Copyright (C) <textyear> <name of author>
```

```
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

Appendix E

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or

XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes

limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example,

statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable

means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

- [1] P. Huber, M. Lindner, and W. Winter, *Comput. Phys. Commun.* **167**, 195 (2005), hep-ph/0407333.
- [2] P. Huber, J. Kopp, M. Lindner, M. Rolinec, and W. Winter, *Comput. Phys. Commun.* **177**, 432 (2007), hep-ph/0701187.
- [3] A. de Gouvea and J. Jenkins, (2008), 0804.3627.
- [4] ISS Physics Working Group, A. Bandyopadhyay *et al.*, (2007), 0710.4947.
- [5] P. Coloma, A. Donini, E. Fernandez-Martinez, and J. Lopez-Pavon, *JHEP* **05**, 050 (2008), 0712.0796.
- [6] A. Gelman and D. Rubin, *Statistical Science* **7**, 457 (1992).